IBM Systems - iSeries

UNIX-Type -- Integrated File System APIs

Version 5 Release 4

IBM Systems - iSeries

UNIX-Type -- Integrated File System APIs

Version 5 Release 4

Note

Before using this information and the product it supports, be sure to read the information in "Notices," on page 553.

Sixth Edition (February 2006)

This edition applies to version 5, release 4, modification 0 of IBM i5/OS (product number 5722-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

© Copyright International Business Machines Corporation 1998, 2006.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Integrated File System APIs 1
APIs 10
access()—Determine File Accessibility. 10
Parameters 10
Authorities 11
Return Value 11
Error Conditions. 11
Error Messages 12
Usage Notes 12
Related Information 13
Example 13
accessx()—Determine File Accessibility for a Class of Users 14
Parameters 14
Authorities 15
Return Value 16
Error Conditions. 16
Error Messages 17
Usage Notes 17
Related Information 18
Example 19
chdir()—Change Current Directory 19
Parameters 19
Authorities 20
Return Value 20
Error Conditions. 20
Error Messages 21
Usage Notes 21
Related Information 22
Example 22
chmod()—Change File Authorizations 22
Parameters 23
Authorities 24
Return Value 25
Error Conditions. 25
Error Messages 26
Usage Notes 26
Related Information 28
Example 28
chown()—Change Owner and Group of File . . . 29
Parameters 29
Authorities 29
Return Value 31
Error Conditions. 31
Error Messages 32
Usage Notes 32
Related Information 33
Example 33
close()—Close File or Socket Descriptor 34
Parameters 34
Authorities 34
Return Value 34
Error Conditions. 34
Error Messages 35
Usage Notes 36
Related Information 36
Example 36
closedir()—Close Directory 37
Parameters 37
Authorities 38
Return Value 38
Error Conditions. 38
Error Messages 39
Usage Notes 39
Related Information 39
Example 40
creat()—Create or Rewrite File 40
Parameters 41
Authorities 41
Return Value 42
Error Conditions. 42
Error Messages 43
Usage Notes 43
Related Information 46
Example 46
creat64()—Create or Rewrite a File (Large File Enabled) 46
Usage Notes 47
DosSetFileLocks()—Lock and Unlock a Byte Range of an Open File 47
Parameters 47
Authorities 48
Return Value 48
Error Conditions. 48
Error Messages 49
Usage Notes 49
Related Information 50
Example 50
DosSetFileLocks64()—Lock and Unlock a Byte Range of an Open File (Large File Enabled). . . . 51
Usage Notes 52
Related Information 52
DosSetRelMaxFH()—Change Maximum Number of File Descriptors 53
Parameters 53
Authorities 53
Return Value 53
Error Conditions. 54
Error Messages 54
Usage Notes 54
Related Information 54
Example 55
dup()—Duplicate Open File Descriptor 55
Parameters 56
Authorities 56
Return Value 56
Error Conditions. 56
Error Messages 56
Usage Notes 57
Related Information 57
Example 57

© Copyright IBM Corp. 1998, 2006

dup2()—Duplicate Open File Descriptor to Another Descriptor . . . 58
Parameters . . . 58
Authorities . . . 58
Return Value . . . 59
Error Conditions. . . . 59
Error Messages . . . 59
Usage Notes . . . 59
Related Information . . . 60
Example . . . 60
faccessx()—Determine File Accessibility for a Class of Users . . . 61
Parameters . . . 61
Authorities . . . 62
Return Value . . . 62
Error Conditions. . . . 63
Error Messages . . . 64
Usage Notes . . . 64
Related Information . . . 65
Example . . . 65
fchdir()—Change Current Directory by Descriptor 66
Parameters . . . 66
Authorities . . . 66
Return Value . . . 66
Error Conditions. . . . 67
Error Messages . . . 67
Usage Notes . . . 68
Related Information . . . 68
Example . . . 68
fchmod()—Change File Authorizations by Descriptor . . . 69
Parameters . . . 69
Authorities . . . 69
Return Value . . . 70
Error Conditions. . . . 70
Error Messages . . . 71
Related Information . . . 71
Example . . . 72
fchown()—Change Owner and Group of File by Descriptor . . . 72
Parameters . . . 73
Authorities . . . 73
Return Value . . . 74
Error Conditions. . . . 74
Error Messages . . . 75
Usage Notes . . . 75
Related Information . . . 76
Example . . . 76
fclear()—Write (Binary Zeros) to Descriptor. . . . 77
Parameters . . . 78
Authorities . . . 78
Return Value . . . 78
Error Conditions. . . . 78
Error Messages . . . 79
Usage Notes . . . 80
Related Information . . . 81
Example . . . 81
fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled) . . . 82
Usage Notes . . . 82
fcntl()—Perform File Control Command . . . 82
Parameters . . . 83
Flags . . . 84
File Locking . . . 85
Authorities . . . 89
Return Value . . . 89
Error Conditions. . . . 89
Error Messages . . . 90
Usage Notes . . . 90
Related Information . . . 91
Example . . . 92
fpathconf()—Get Configurable Path Name Variables by Descriptor . . . 92
Parameters . . . 93
Authorities . . . 93
Return Value . . . 93
Error Conditions. . . . 93
Error Messages . . . 94
Usage Notes . . . 94
Related Information . . . 95
Example . . . 95
fstat()—Get File Information by Descriptor . . . 95
Parameters . . . 96
Authorities . . . 96
Return Value . . . 96
Error Conditions. . . . 96
Error Messages . . . 97
Usage Notes . . . 97
Related Information . . . 98
Example . . . 99
fstat64()—Get File Information by Descriptor (Large File Enabled) . . . 99
Usage Notes. . . . 100
Example . . . 100
fstatvfs()—Get File System Information by Descriptor . . . 101
Parameters . . . 101
Return Value . . . 102
Error Conditions . . . 102
Error Messages . . . 103
Usage Notes. . . . 103
Related Information . . . 104
Example . . . 104
fstatvfs64()—Get File System Information by Descriptor (64-Bit Enabled) . . . 105
Usage Notes. . . . 105
fsync()—Synchronize Changes to File . . . 106
Parameters . . . 106
Authorities . . . 106
Return Value . . . 106
Error Conditions . . . 106
Error Messages . . . 107
Usage Notes. . . . 107
Related Information . . . 108
Example . . . 108
ftruncate()—Truncate File . . . 109
Parameters . . . 109
Authorities . . . 109
Return Value . . . 109
Error Conditions . . . 109
Error Messages . . . 111
Usage Notes . . . 111

Related Information 112
Example 112
ftruncate64()—Truncate File (Large File Enabled) 113
Usage Notes. 113
getcwd()—Get Current Directory 113
Parameters 114
Authorities 114
Return Value 114
Error Conditions 114
Error Messages 116
Usage Notes. 116
Related Information 116
Example 116
getegid()—Get Effective Group ID 117
Parameters 117
Authorities 117
Return Value 117
Error Conditions 117
Related Information 118
Example 118
geteuid()—Get Effective User ID 118
Parameters 118
Authorities 118
Return Value 119
Error Conditions 119
Related Information 119
Example 119
getgid()—Get Real Group ID 119
Parameters 120
Authorities 120
Return Value 120
Error Conditions 120
Related Information 120
Example 120
getgrgid()—Get Group Information Using Group ID 121
Parameters 121
Authorities 121
Return Value 121
Error Conditions 121
Related Information 122
Example 122
getgrgid_r()—Get Group Information Using Group ID 122
Parameters 123
Authorities 123
Return Value 123
Error Conditions 123
Related Information 123
Example 124
getgrgid_r_ts64()—Get Group Information Using Group ID. 124
getgrnam()—Get Group Information Using Group Name 125
Parameters 125
Authorities 125
Return Value 125
Error Conditions 126
Related Information 126
Example 126
getgrnam_r()—Get Group Information Using Group Name 127
Parameters 127
Authorities 127
Return Value 127
Error Conditions 127
Related Information 128
Example 128
getgrnam_r_ts64()—Get Group Information Using Group Name 129
getgroups()—Get Group IDs 129
Parameters 129
Authorities 130
Return Value 130
Error Conditions 130
Usage Notes. 130
Related Information 130
getpwnam()—Get User Information for User Name 131
Parameters 131
Authorities 131
Return Value 131
Error Conditions 132
Usage Notes. 132
Related Information 132
Example 132
getpwnam_r()—Get User Information for User Name 133
Parameters 133
Authorities 133
Return Value 134
Error Conditions 134
Usage Notes. 134
Related Information 134
Example 134
getpwnam_r_ts64()—Get User Information for User Name 135
getpwuid()—Get User Information for User ID . . 135
Parameters 136
Authorities 136
Return Value 136
Error Conditions 136
Usage Notes. 136
Related Information 137
Example 137
getpwuid_r()—Get User Information for User ID 137
Parameters 138
Authorities 138
Return Value 138
Error Conditions 138
Usage Notes. 139
Related Information 139
Example 139
getpwuid_r_ts64()—Get User Information for User ID 140
getuid()—Get Real User ID 140
Parameters 140
Authorities 140
Return Value 141
Error Conditions 141
Related Information 141
Example 141

ioctl()—Perform I/O Control Request 141
Parameters 142
Authorities 146
Return Value 147
Error Conditions 147
Error Messages 148
Usage Notes. 148
Related Information 148
lchown()—Change Owner and Group of Symbolic Link 149
Parameters 149
Authorities 149
Return Value 151
Error Conditions 151
Error Messages 152
Usage Notes. 152
Related Information 152
Example 153
link()—Create Link to File 153
Parameters 154
Authorities 154
Return Value 155
Error Conditions 155
Error Messages 156
Usage Notes. 156
Related Information 156
Example 157
lseek()—Set File Read/Write Offset 157
Parameters 158
Authorities 159
Return Value 159
Error Conditions 159
Error Messages 160
Usage Notes. 160
Related Information 161
Example 161
lseek64()—Set File Read/Write Offset (Large File Enabled) 161
Usage Notes. 162
lstat()—Get File or Link Information. 162
Parameters 163
Authorities 163
Return Value 163
Error Conditions 163
Error Messages 164
Usage Notes. 165
Related Information 166
Example 166
lstat64()—Get File or Link Information (Large File Enabled) 167
Usage Notes. 168
Example 168
mkdir()—Make Directory 169
Parameters 169
Authorities 170
Return Value 170
Error Conditions 170
Error Messages 171
Usage Notes. 172
Related Information 174
Example 174
mkfifo()—Make FIFO Special File. 175
Parameters 175
Authorities 176
Return Value 176
Error Conditions 176
Error Messages 177
Usage Notes. 177
Related Information 178
Example 178
mmap()—Memory Map a File 179
Parameters 180
Authorities 181
Return Value 182
Error Conditions 182
Error Messages 182
Usage Notes. 183
Related Information 183
Example 184
mmap64()—Memory map a Stream File (Large File Enabled) 186
Usage Notes. 186
mprotect()—Change Access Protection for Memory Mapping 186
Parameters 187
Authorities 187
Return Value 187
Error Conditions 187
Error Messages 188
Usage Notes. 188
Related Information 188
Example 189
msync()—Synchronize Modified Data with Mapped File. 190
Parameters 190
Authorities 191
Return Value 191
Error Conditions 191
Error Messages 191
Usage Notes. 191
Related Information 192
Example 192
munmap()—Remove Memory Mapping 193
Parameters 193
Authorities 194
Return Value 194
Error Conditions 194
Error Messages 194
Usage Notes. 194
Related Information 194
Example 195
open()—Open File 195
Parameters 196
Using the oflag Parameter 197
Using CCSIDs and code pages 201
Authorities 202
Return Value 203
Error Conditions 203
Error Messages 205
Usage Notes. 205
Related Information 209
Examples. 209

open64()—Open File (Large File Enabled) 211
Usage Notes. 212
opendir()—Open Directory 212
Parameters 212
Authorities 213
Return Value 213
Error Conditions 213
Error Messages 214
Usage Notes. 215
Related Information 215
Example 216
pathconf()—Get Configurable Path Name Variables 216
Parameters 217
Authorities 218
Return Value 218
Error Conditions 219
Error Messages 220
Usage Notes. 220
Related Information 220
Example 220
pipe()—Create an Interprocess Channel. 221
Parameters 221
Authorities 221
Return Value 222
Error Conditions 222
Usage Notes. 222
Related Information 222
Example 222
pread()—Read from Descriptor with Offset . . . 223
Parameters 224
Authorities 224
Return Value 224
Error Conditions 224
Error Messages 225
Usage Notes. 225
Related Information 226
Example 227
pread64()—Read from Descriptor with Offset (large file enabled) 228
Usage Notes. 228
Example 228
pwrite()—Write to Descriptor with Offset 229
Parameters 229
Authorities 230
Return Value 230
Error Conditions 230
Error Messages 231
Usage Notes. 231
Related Information 233
Example 233
pwrite64()—Write to Descriptor with Offset (large file enabled) 234
Usage Notes. 235
Example 235
QlgAccess()—Determine File Accessibility (using NLS-enabled path name) 235
Parameters 236
Related Information 236
Example 236
QlgAccessx()—Determine File Accessibility for a Class of Users (using NLS-enabled path name) . . 237
Parameters 238
Related Information 238
Example 238
QlgChdir()—Change Current Directory (using NLS-enabled path name) 239
Parameters 239
Related Information 239
Example 239
QlgChmod()—Change File Authorizations (using NLS-enabled path name) 240
Parameters 241
Related Information 241
Example 241
QlgChown()—Change Owner and Group of File (using NLS-enabled path name) 242
Parameters 242
Related Information 242
Example 243
QlgCreat()—Create or Rewrite File (using NLS-enabled path name) 244
Parameters 244
Related Information 244
Example 244
QlgCreat64()—Create or Rewrite a File (large file enabled and using NLS-enabled path name) . . . 245
Parameters 246
Related Information 246
Example 246
QlgCvtPathToQSYSObjName()— Resolve Integrated File System Path Name into QSYS Object Name (using NLS-enabled path name) . . 247
QlgGetAttr()—Get Attributes (using NLS-enabled path name) 247
QlgGetcwd()—Get Current Directory (using NLS-enabled path name) 248
Parameters 248
Related Information 248
Example 248
QlgGetPathFromFileID()—Get Path Name of Object from Its File ID (using NLS-enabled path name). . 249
Parameters 250
Related Information 250
Example 250
QlgGetpwnam()—Get User Information for User Name (using NLS-enabled path name) 252
Parameters 252
Authorities 253
Return Value 253
Error Conditions 253
Usage Notes. 253
Related Information 253
Example 253
QlgGetpwnam_r()—Get User Information for User Name (using NLS-enabled path name) 254
Parameters 254
Authorities 255
Return Value 255
Error Conditions 255
Usage Notes. 255
Related Information 255
Example 256

QlgGetpwuid()—Get User Information for User ID (using NLS-enabled path name) 256
Parameters 257
Authorities 257
Return Value 257
Error Conditions 257
Usage Notes. 257
Related Information 258
Example 258
QlgGetpwuid_r()—Get User Information for User ID (using NLS-enabled path name) 258
Parameters 259
Authorities 259
Return Value 259
Error Conditions 259
Usage Notes. 260
Related Information 260
Example 260
QlgLchown()—Change Owner and Group of Symbolic Link (using NLS-enabled path name) . . 261
Parameters 261
Related Information 261
Example 261
QlgLink()—Create Link to File (using NLS-enabled path name) 263
Parameters 263
Related Information 263
Example 263
QlgLstat()—Get File or Link Information (using NLS-enabled path name) 265
Parameters 265
Related Information 265
Example 266
QlgLstat64()—Get File or Link Information (large file enabled and using NLS-enabled path name). . 267
Parameters 268
Related Information 268
Example 268
QlgMkdir()—Make Directory (using NLS-enabled path name) 270
Parameters 270
Related Information 270
Example 270
QlgMkfifo()—Make FIFO Special File (using NLS-enabled path name) 271
Parameters 272
Related Information 272
Example 272
QlgOpen()—Open a File (using NLS-enabled path name) 273
Parameters 273
Related Information 273
Example 273
QlgOpen64()—Open File (large file enabled and using NLS-enabled path name) 274
Parameters 275
Related Information 275
QlgOpendir()—Open Directory (using NLS-enabled path name) 275
Parameters 275
Related Information 275
Example 276
QlgPathconf()—Get Configurable Path Name Variables (using NLS-enabled path name) 278
Parameters 278
Related Information 278
Example 278
QlgProcessSubtree()—Process a Path Name (using NLS-enabled path name) 279
QlgReaddir()—Read Directory Entry (using NLS-enabled path name) 280
Parameters 280
Related Information 281
Example 281
QlgReaddir_r()—Read Directory Entry (using NLS-enabled path name) 282
Parameters 282
Related Information 283
Example 283
QlgReadlink()—Read Value of Symbolic Link (using NLS-enabled path name) 284
Parameters 285
Related Information 285
Example 285
QlgRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists (using NLS-enabled path name) . 286
Parameters 287
Related Information 287
Example 287
QlgRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists (using NLS-enabled path name) 288
Parameters 289
Related Information 289
Example 289
QlgRmdir()—Remove Directory (using NLS-enabled path name) 290
Parameters 290
Related Information 291
Example 291
QlgSaveStgFree()—Save Storage Free (using NLS-enabled path name) 292
QlgSetAttr()—Set Attributes (using NLS-enabled path name) 292
QlgStat()—Get File Information (using NLS-enabled path name) 293
Parameters 293
Related Information 293
Example 294
QlgStat64()—Get File Information (large file enabled and using NLS-enabled path name) . . . 295
Parameters 295
Related Information 295
Example 295
QlgStatvfs()—Get File System Information (using NLS-enabled path name) 296
Parameters 297
Related Information 297
Example 297
QlgStatvfs64()—Get File System Information (64-Bit enabled and using NLS-enabled path name) . . . 298
Parameters 299

Related Information 299
Example 299
QlgSymlink()—Make Symbolic Link (using NLS-enabled path name) 300
Parameters 300
Related Information 300
Example 301
QlgUnlink()—Remove Link to File (using NLS-enabled path name) 302
Parameters 302
Related Information 302
Example 303
QlgUtime()—Set File Access and Modification Times (using NLS-enabled path name) 303
Parameters 304
Related Information 304
Example 304
Perform Miscellaneous File System Functions (QP0FPTOS) API 305
Authorities and Locks 305
Required Parameter Group 306
Usage Notes. 308
Error Messages 308
Examples. 308
Qp0lCvtPathToQSYSObjName()— Resolve Integrated File System Path Name into QSYS Object Name 308
Parameters 309
Authorities 310
Returned Data Format 310
Field Descriptions 311
Error Conditions 311
Error Messages 311
Usage Notes. 312
Related Information 313
Example 313
Perform File System Operation (QP0LFLOP) API 315
Authorities and Locks 315
Required Parameter Group 316
Output Buffer Description 318
FLOP0100 Structure Description 318
FLOP0300 Output Structure Description . . . 318
FLOP0400 Output Structure Description . . . 319
Input Buffer Description. 320
Format of FLOP0200 Structure. 320
Format of FLOP0300 Input Structure 321
Format of FLOP0400 Input Structure 321
Field Descriptions 321
Usage Notes. 325
Error Messages 325
Qp0lGetAttr()—Get Attributes 326
Parameters 326
Authorities 345
Return Value 345
Error Conditions 345
Error Messages 346
Usage Notes. 347
Related Information 347
Example 348
Qp0lGetPathFromFileID()—Get Path Name of Object from Its File ID 351
Parameters 352
Authorities 352
Return Value 352
Error Conditions 353
Error Messages 353
Usage Notes. 353
Related Information 354
Example 354
Qp0lOpen()—Open File 354
Parameters 355
Related Information 355
Example 355
Qp0lProcessSubtree()—Process a Path Name . . . 356
Parameters 356
Authorities 360
Return Value 361
Error Conditions 361
Error Messages 361
Usage Notes. 362
Scenarios 364
Figure: Directory Structure A 365
Figure: Directory Structure B 366
Scenario 1 366
Figure: Scenario 1 API Input 366
Figure: Results of a call 367
Scenario 2 367
Figure: Scenario 2 API Input 367
Figure: Results of a call 368
Scenario 3 368
Figure: Scenario 3 API Input 368
Figure: Results of a call 369
Scenario 4 369
Figure: Scenario 4 API Input 369
Figure: Results of a call 369
Related Information 370
Example 370
Qp0lRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists 373
Parameters 374
Authorities 374
Return Value 375
Error Conditions 376
Error Messages 377
Usage Notes. 377
Related Information 379
Example 379
Qp0lRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists. 379
Parameters 380
Authorities 380
Return Value 382
Error Conditions 382
Error Messages 383
Usage Notes. 383
Related Information 385
Example 385
Retrieve Object References (QP0LROR) 385
Parameters 386
Authorities and Locks 387
Output Structure Formats 387

RORO0100 Output Format Description (*Qp0l_RORO0100_Output*) 387
RORO0200 Output Format Description (*Qp0l_RORO0200_Output*) 387
Job Using Object Structure Description (*Qp0l_Job_Using_Object*) 388
Simple Object Reference Types Structure Description (*Qp0l_Sim_Ref_Types_Output*) . . . 389
Extended Object Reference Types Structure Description (*Qp0l_Ext_Ref_Types_Output*) . . . 389
iSeries NetServer Session Using Object Structure Description (*Qp0l_Session_Using_Object Structure*) 391
Field Descriptions for RORO0100 and RORO0200 Output Structures and their Imbedded Structures 391
Error Messages 394
Usage Notes. 395
Related Information 396
Example 396
Qp0lSaveStgFree()—Save Storage Free 399
Parameters 400
Authorities 401
Return Value 401
Error Conditions 401
Error Messages 402
Usage Notes. 402
Related Information 403
Example 403
Qp0lSetAttr()—Set Attributes 403
Parameters 404
Authorities 409
Return Value 410
Error Conditions 410
Error Messages 412
Usage Notes. 412
Related Information 414
Example 414
Qp0lUnlink()—Remove Link to File 418
Parameters 418
Related Information 418
Example 418
Qp0zPipe()—Create Interprocess Channel with Sockets 419
Parameters 419
Authorities 419
Return Value 419
Error Conditions 420
Usage Notes. 420
Related Information 420
qsygetgroups()—Get Supplemental Group IDs . . 420
Parameters 421
Authorities 421
Return Value 421
Error Conditions 421
qsysetegid()—Set Effective Group ID 421
Parameters 422
Authorities and Locks 422
Return Value 422
Error Conditions 422
qsyseteuid()—Set Effective User ID 423
Parameters 423
Authorities and Locks 423
Return Value 423
Error Conditions 423
qsysetgid()—Set Group ID 424
Parameters 424
Authorities and Locks 424
Return Value 425
Error Conditions 425
qsysetgroups()—Set Supplemental Group IDs . . 425
Parameters 426
Authorities and locks. 426
Return Value 426
Error Conditions 426
qsysetregid()—Set Real and Effective Group IDs 427
Parameters 427
Authorities and Locks 428
Return Value 428
Error Conditions 428
qsysetreuid()—Set Real and Effective User IDs . . 428
Parameters 429
Authorities and Locks 429
Return Value 429
Error Conditions 429
qsysetuid()—Set User ID 430
Parameters 430
Authorities and Locks 430
Return Value 431
Error Conditions 431
Retrieve Network File System Export Entries (QZNFRTVE) API 431
Authorities and Locks 431
Usage Notes. 432
Required Parameter Group 432
Receiver Variable Description 433
EXPE0100 and EXPE0200 format 433
Returned Records Feedback Information Description 434
Format of Returned Records Feedback Information 435
Field Descriptions 435
Error Messages 436
read()—Read from Descriptor 437
Parameters 438
Authorities 438
Return Value 438
Error Conditions 438
Error Messages 440
Usage Notes. 440
Related Information 442
Example 442
readdir()—Read Directory Entry 443
Parameters 443
Authorities 444
Return Value 444
Error Conditions 444
Error Messages 445
Usage Notes. 446
Related Information 446
Example 446
readdir_r()—Read Directory Entry 447

Parameters 447
Authorities 448
Return Value 448
Error Conditions 448
Error Messages 449
Usage Notes. 450
Related Information 450
Example 451
readdir_r_ts64()—Read Directory Entry. 451
readlink()—Read Value of Symbolic Link 452
Parameters 452
Authorities 452
Return Value 453
Error Conditions 453
Error Messages 454
Usage Notes. 454
Related Information 455
Example 455
readv()—Read from Descriptor Using Multiple Buffers 455
Parameters 456
Authorities 456
Return Value 456
Error Conditions 456
Error Messages 458
Usage Notes. 458
Related Information 459
rename()—Rename File or Directory. 460
Authorities and Locks 460
Parameters 460
Usage Notes. 461
Related Information 461
rewinddir()—Reset Directory Stream to Beginning 461
Parameters 462
Authorities 462
Return Value 462
Error Conditions 462
Error Messages 462
Usage Notes. 462
Related Information 462
Example 462
rmdir()—Remove Directory. 463
Parameters 464
Authorities 464
Return Value 465
Error Conditions 465
Error Messages 466
Usage Notes. 466
Related Information 467
Example 467
stat()—Get File Information. 468
Parameters 468
Authorities 470
Return Value 471
Error Conditions 471
Error Messages 472
Usage Notes. 472
Related Information 473
Example 474
stat64()—Get File Information (Large File Enabled) 475
Parameters 475
Usage Notes. 477
statvfs()—Get File System Information 478
Parameters 478
Authorities 480
Return Value 480
Error Conditions 480
Error Messages 481
Usage Notes. 481
Related Information 482
Example 482
statvfs64()—Get File System Information (64-Bit Enabled) 483
Parameters 483
Usage Notes. 485
symlink()—Make Symbolic Link 485
Parameters 486
Authorities 486
Return Value 486
Error Conditions 486
Error Messages 487
Usage Notes. 487
Related Information 488
Example 488
sysconf()—Get System Configuration Variables . . 488
Parameters 489
Authorities 490
Return Value 490
Error Conditions 490
Error Messages 490
Related Information 490
Example 490
umask()—Set Authorization Mask for Job 491
Parameters 491
Authorities 491
Return Value 491
Error Conditions 491
Error Messages 491
Usage Notes. 492
Related Information 492
Example 492
unlink()—Remove Link to File. 492
Parameters 493
Authorities 493
Return Value 494
Error Conditions 494
Error Messages 496
Usage Notes. 496
Related Information 497
Example 497
utime()—Set File Access and Modification Times 497
Parameters 498
Authorities 498
Return Value 498
Error Conditions 499
Error Messages 500
Usage Notes. 500
Related Information 501
Example 501
write()—Write to Descriptor 502
Parameters 503
Authorities 503

Return Value 503
Error Conditions 504
Error Messages 505
Usage Notes. 506
Related Information 507
Example 508
writev()—Write to Descriptor Using Multiple Buffers 509
Parameters 509
Authorities 509
Return Value 509
Error Conditions 509
Error Messages 511
Usage Notes. 511
Related Information 512
Exit Programs 513
Integrated File System Scan on Close Exit Program 513
Restrictions 514
Authorities and Locks 515
Program Data 515
Required Parameter Group 515
Format of Integrated File System Close Exit Information (Input) 515
Format of Status Information (Output) 516
Field Descriptions 516
Usage Notes. 521
Related Information 522
Integrated File System Scan on Open Exit Program 523
Restrictions 524
Authorities and Locks 524
Program Data 524
Required Parameter Group 525
Format of Integrated File System Open Exit Information (Input) 525
Format of Status Information (Output) 525
Field Descriptions 526
Scan Key List and Scan Key Signatures. . . . 530
Coded Character Set Identifier (CCSID) Information 531
Usage Notes. 532
Related Information 533
Process a Path Name Exit Program 533
Authorities and Locks 534
Parameters 534
Save Storage Free Exit Program 535
Authorities and Locks 535
Required Parameter Group 536
Related Information 536
Concepts 537
Header Files for UNIX-Type Functions 537
Errno Values for UNIX-Type Functions 539
Integrated File System APIs—Time Stamp Updates 548

Appendix. Notices 553
Programming Interface Information 554
Trademarks 555
Terms and Conditions 556

Integrated File System APIs

The integrated file system is a part of i5/OS(TM) that supports stream input/output and storage management similar to personal computer and UNIX(R) operating systems while providing an integrating structure over all information stored in your server.

The stream file support is designed for efficient use in client/server applications. Stream files are particularly well suited for storing long continuous strings of data such as the text of documents, images, audio, and video.

The integrated file system provides a hierarchical directory structure that supports UNIX-based open system standards, such as Portable Operating System Interface for Computer Environments (POSIX)** and The Single UNIX(R); Specification. This file and directory structure provides the users of PC operating systems with a familiar environment.

In addition to providing an interface for users and application to access stream files, the integrated file system also provides a common interface to access database files, documents and other objects stored on the server.

For more information, see the Integrated file system information in the Files and file systems topic.

The integrated file system APIs are:

- "access()—Determine File Accessibility" on page 10 (Determine file accessibility) determines whether a file can be accessed in a particular manner.
- "accessx()—Determine File Accessibility for a Class of Users" on page 14 (Determine File Accessibility for a Class of Users) determines whether a file can be accessed by a specified class of users in a particular manner.
- "chdir()—Change Current Directory" on page 19 (Change current directory) makes the directory named by path the new current directory.
- "chmod()—Change File Authorizations" on page 22 (Change file authorizations) changes the mode of the file or directory specified in path.
- "chown()—Change Owner and Group of File" on page 29 (Change owner and group of file) changes the owner and group of a file.
- "close()—Close File or Socket Descriptor" on page 34 (Close file descriptor) closes a descriptor, fildes.
- "closedir()—Close Directory" on page 37 (Close directory) closes the directory stream indicated by dirp.
- "creat()—Create or Rewrite File" on page 40 (Create new file or rewrite existing file) creates a new file or rewrites an existing file so that it is truncated to zero length.
- "creat64()—Create or Rewrite a File (Large File Enabled)" on page 46 (Create new file or rewrite existing file (large file enabled)) creates a new file or rewrites an existing file so that it is truncated to zero length.
- "DosSetFileLocks()—Lock and Unlock a Byte Range of an Open File" on page 47 (Lock and unlock a range of an open file) locks and unlocks a range of an open file.
- "DosSetFileLocks64()—Lock and Unlock a Byte Range of an Open File (Large File Enabled)" on page 51 (Lock and unlock a range of an open file (large file enabled)) locks and unlocks a range of an open file.
- "DosSetRelMaxFH()—Change Maximum Number of File Descriptors" on page 53 (Change maximum number of file descriptors) requests that the system change the maximum number of file descriptors for the calling process (job).
- "dup()—Duplicate Open File Descriptor" on page 55 (Duplicate open file descriptor) returns a new open file descriptor.

© Copyright IBM Corp. 1998, 2006

- "dup2()—Duplicate Open File Descriptor to Another Descriptor" on page 58 (Duplicate open file descriptor to another descriptor) returns a descriptor with the value fildes2.
- "faccessx()—Determine File Accessibility for a Class of Users" on page 61 (Determine File Accessibility for a Class of Users) determines whether a file can be accessed by a specified class of users in a particular manner.
- "fchdir()—Change Current Directory by Descriptor" on page 66 (Change Current Directory by Descriptor) makes the directory named by fildes the new current directory.
- "fchmod()—Change File Authorizations by Descriptor" on page 69 (Change file authorizations by descriptor) sets the file permission bits of the open file identified by fildes, its file descriptor.
- "fchown()—Change Owner and Group of File by Descriptor" on page 72 (Change owner and group of file by descriptor) changes the owner and group of a file.
- "fclear()—Write (Binary Zeros) to Descriptor" on page 77 (Write (Binary Zeros) to Descriptor) clears a file.
- "fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)" on page 82 (Write (Binary Zeros) to Descriptor (Large File Enabled)) clears a file.
- "fcntl()—Perform File Control Command" on page 82 (Perform file control command) performs various actions on open descriptors.
- "fpathconf()—Get Configurable Path Name Variables by Descriptor" on page 92 (Get configurable path name variables by descriptor) determines the value of a configuration variable (name) associated with a particular file descriptor (file_descriptor).
- "fstat()—Get File Information by Descriptor" on page 95 (Get file information by descriptor) gets status information about the file specified by the open file descriptor file_descriptor and stores the information in the area of memory indicated by the buf argument.
- "fstat64()—Get File Information by Descriptor (Large File Enabled)" on page 99 (Get file information by descriptor (large file enabled)) gets status information about the file specified by the open file descriptor file_descriptor and stores the information in the area of memory indicated by the buf argument.
- "fstatvfs()—Get File System Information by Descriptor" on page 101 (Get File System Information by Descriptor) gets status information about the file system that contains the file referenced by the open file descriptor fildes.
- "fstatvfs64()—Get File System Information by Descriptor (64-Bit Enabled)" on page 105 (Get file system information by descriptor (64-bit enabled)) gets status information about the file system that contains the file referred to by the open file descriptor fildes.
- "fsync()—Synchronize Changes to File" on page 106 (Synchronize changes to file) transfers all data for the file indicated by the open file descriptor file_descriptor to the storage device associated with file_descriptor.
- "ftruncate()—Truncate File" on page 109 (Truncate file) truncates the file indicated by the open file descriptor file_descriptor to the indicated length.
- "ftruncate64()—Truncate File (Large File Enabled)" on page 113 (Truncate file (large file enabled)) truncates the file indicated by the open file descriptor file_descriptor to the indicated length.
- "getcwd()—Get Current Directory" on page 113 (Get Current Directory) determines the absolute path name of the current directory and stores it in buf.
- "getegid()—Get Effective Group ID" on page 117 (Get effective group ID) returns the effective group ID (gid) of the calling thread.
- "geteuid()—Get Effective User ID" on page 118 (Get effective user ID) returns the effective user ID (uid) of the calling thread.
- "getgid()—Get Real Group ID" on page 119 (Get real group ID) returns the real group ID (gid) of the calling thread.
- "getgrgid()—Get Group Information Using Group ID" on page 121 (Get group information using group ID) returns a pointer to an object of type struct group containing an entry from the user database with a matching gid.

- “getgrgid_r()—Get Group Information Using Group ID” on page 122 (Get group information using group ID) updates the group structure pointed to by grp and stores a pointer to that structure in the location pointed to by result.
- “getgrgid_r_ts64()—Get Group Information Using Group ID” on page 124 (Get group information using group ID) updates the group structure pointed to by grp and stores a pointer to that structure in the location pointed to by result.
- “getgrnam()—Get Group Information Using Group Name” on page 125 (Get group information using group name) returns a pointer to an object of type struct group containing an entry from the user database with a matching name.
- “getgrnam_r()—Get Group Information Using Group Name” on page 127 (Get group information using group name) updates the group structure pointed to by grp and stores a pointer to that structure in the location pointed to by result.
- “getgrnam_r_ts64()—Get Group Information Using Group Name” on page 129 (Get group information using group name) updates the group structure pointed to by grp and stores a pointer to that structure in the location pointed to by result.
- “getgroups()—Get Group IDs” on page 129 (Get group IDs) returns the number of primary and supplementary group IDs associated with the calling thread without modifying the array pointed to by the grouplist argument.
- “getpwnam()—Get User Information for User Name” on page 131 (Get user information for user name) returns a pointer to an object of type struct passwd containing an entry from the user database with a matching name.
- “getpwnam_r()—Get User Information for User Name” on page 133 (Get User Information for User Name) updates the*passwd*structure pointed to by*pwd*and stores a pointer to that structure in the location pointed to by*result*.
- “getpwnam_r_ts64()—Get User Information for User Name” on page 135 (Get user information for user name) updates the passwd structure pointed to by pwd and stores a pointer to that structure in the location pointed to by result.
- “getpwuid()—Get User Information for User ID” on page 135 (Get user information for user ID) returns a pointer to an object of type struct passwd containing an entry from the user database with a matching uid.
- “getpwuid_r()—Get User Information for User ID” on page 137 (Get User Information for User ID) updates the*passwd*structure pointed to by*pwd*and stores a pointer to that structure in the location pointed to by*result*.
- “getpwuid_r_ts64()—Get User Information for User ID” on page 140 (Get user information for user ID) updates the passwd structure pointed to by pwd and stores a pointer to that structure in the location pointed to by result.
- “getuid()—Get Real User ID” on page 140 (Get real user ID) returns the real user ID (uid) of the calling thread.
- “ioctl()—Perform I/O Control Request” on page 141 (Perform I/O control request) performs control functions (requests) on a file descriptor.
- “lchown()—Change Owner and Group of Symbolic Link” on page 149 (Change owner and group of symbolic link) changes the owner and group of a file. If the named file is a symbolic link, lchown() changes the owner or group of the link itself rather than the object to which the link points.
- “link()—Create Link to File” on page 153 (Create link to file) provides an alternative path name for the existing file, so that the file can be accessed by either the existing name or the new name.
- “lseek()—Set File Read/Write Offset” on page 157 (Set file read/write offset) changes the current file offset to a new position in the file.
- “lseek64()—Set File Read/Write Offset (Large File Enabled)” on page 161 (Set file read/write offset (large file enabled)) changes the current file offset to a new position in the file.
- “lstat()—Get File or Link Information” on page 162 (Get file or link information) gets status information about a specified file and places it in the area of memory pointed to by buf.

- "lstat64()—Get File or Link Information (Large File Enabled)" on page 167 (Get file or link information (large file enabled)) gets status information about a specified file and places it in the area of memory pointed to by buf.
- "mkdir()—Make Directory" on page 169 (Make directory) creates a new, empty directory whose name is defined by path.
- "mkfifo()—Make FIFO Special File" on page 175 (Make FIFO special file) creates a new FIFO special file (FIFO) whose name is defined by path.
- "mmap()—Memory Map a File" on page 179 (Memory map a file) establishes a mapping between a process' address space and a stream file.
- "mmap64()—Memory map a Stream File (Large File Enabled)" on page 186 (Memory map a stream file (large file enabled)) is used to establish a memory mapping of a file.
- "mprotect()—Change Access Protection for Memory Mapping" on page 186 (Change access protection for memory mapping) is used to change the access protection of a memory mapping to that specified by protection.
- "msync()—Synchronize Modified Data with Mapped File" on page 190 (Synchronize modified data with mapped file) can be used to write modified data from a shared mapping (created using the mmap() function) to non-volatile storage or invalidate privately mapped pages.
- "munmap()—Remove Memory Mapping" on page 193 (Remove memory mapping) removes addressability to a range of memory mapped pages of a process's address space.
- "open()—Open File" on page 195 (Open file) opens a file and returns a number called a file descriptor.
- "open64()—Open File (Large File Enabled)" on page 211 (Open file (large file enabled)) opens a file and returns a number called a file descriptor.
- "opendir()—Open Directory" on page 212 (Open directory) opens a directory so that it can be read with the readdir() function.
- "pathconf()—Get Configurable Path Name Variables" on page 216 (Get configurable path name variables) lets an application determine the value of a configuration variable (name) associated with a particular file or directory (path).
- "pipe()—Create an Interprocess Channel" on page 221 (Create interprocess channel) creates a data pipe and places two file descriptors, one each into the arguments fildes[0] and fildes[1], that refer to the open file descriptions for the read and write ends of the pipe, respectively.
- "pread()—Read from Descriptor with Offset" on page 223 (Read from Descriptor with Offset) reads*nbyte*bytes of input into the memory area indicated by*buf*.
- "pread64()—Read from Descriptor with Offset (large file enabled)" on page 228 (Read from Descriptor with Offset (large file enabled)) reads*nbyte*bytes of input into the memory area indicated by*buf*.
- "pwrite()—Write to Descriptor with Offset" on page 229 (Write to Descriptor with Offset) writes*nbyte*bytes from*buf*to the file associated with*file_descriptor*.
- "pwrite64()—Write to Descriptor with Offset (large file enabled)" on page 234 (Write to Descriptor with Offset (large file enabled)) writes*nbyte*bytes from*buf*to the file associated with*file_descriptor*.
- "QlgAccess()—Determine File Accessibility (using NLS-enabled path name)" on page 235 (Determine file accessibility (using NLS-enabled path name)) determines whether a file can be accessed in a particular manner.
- "QlgAccessx()—Determine File Accessibility for a Class of Users (using NLS-enabled path name)" on page 237 (Determine File Accessibility for a Class of Users (using NLS-enabled path name)) determines whether a file can be accessed in a particular manner by a specified class of users.
- "QlgChdir()—Change Current Directory (using NLS-enabled path name)" on page 239 (Change current directory (using NLS-enabled path name)) makes the directory named by path the new current directory.
- "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240 (Change file authorizations (using NLS-enabled path name)) changes the mode of the file or directory specified in path.

- “QlgChown()—Change Owner and Group of File (using NLS-enabled path name)” on page 242 (Change owner and group of file (using NLS-enabled path name)) changes the owner and group of a file.
- “QlgCreat()—Create or Rewrite File (using NLS-enabled path name)” on page 244 (Create or rewrite file (using NLS-enabled path name)) creates a new file or rewrites an existing file so that it is truncated to zero length.
- “QlgCreat64()—Create or Rewrite a File (large file enabled and using NLS-enabled path name)” on page 245 (Create or rewrite a file (large file enabled and using NLS-enabled path name)) creates a new file or rewrites an existing file so that it is truncated to zero length.
- “QlgCvtPathToQSYSObjName()— Resolve Integrated File System Path Name into QSYS Object Name (using NLS-enabled path name)” on page 247 (Resolve integrated file system path name into QSYS object name (using NLS-enabled path name)) resolves a given integrated file system path name into the three-part QSYS.LIB file system name: library, object, and member.
- “QlgGetAttr()—Get Attributes (using NLS-enabled path name)” on page 247 (Get attributes (using NLS-enabled path name)) gets one or more attributes, on a single call, for the object that is referred to by the input Path_Name.
- “QlgGetcwd()—Get Current Directory (using NLS-enabled path name)” on page 248 (Get current directory (using NLS-enabled path name)) determines the absolute path name of the current directory and returns a pointer to it.
- “QlgGetPathFromFileID()—Get Path Name of Object from Its File ID (using NLS-enabled path name)” on page 249 (Get path name of object from its file ID (using NLS-enabled path name)) determines an absolute path name of the file identified by fileid and stores it in buf.
- “QlgGetpwnam()—Get User Information for User Name (using NLS-enabled path name)” on page 252 (Get user information for user name (using NLS-enabled path name)) returns a pointer to an object of type struct qplg_passwd containing an entry from the user database with a matching name.
- “QlgGetpwnam_r()—Get User Information for User Name (using NLS-enabled path name)” on page 254 (Get user information for user name (using NLS-enabled path name)) updates the qplg_passwd structure pointed to by pwd and stores a pointer to that structure in the location pointed to by result.
- “QlgGetpwuid()—Get User Information for User ID (using NLS-enabled path name)” on page 256 (Get user information for user ID (using NLS-enabled path name)) returns a pointer to an object of type struct qplg_passwd containing an entry from the user database with a matching user ID (UID).
- “QlgGetpwuid_r()—Get User Information for User ID (using NLS-enabled path name)” on page 258 (Get user information for user ID (using NLS-enabled path name)) updates the qplg_passwd structure pointed to by pwd and stores a pointer to that structure in the location pointed to by result.
- “QlgLchown()—Change Owner and Group of Symbolic Link (using NLS-enabled path name)” on page 261 (Change owner and group of symbolic link (using NLS-enabled path name)) changes the owner and group of a file.
- “QlgLink()—Create Link to File (using NLS-enabled path name)” on page 263 (Create link to file (using NLS-enabled path name)) provides an alternative path name for the existing file so that the file can be accessed by either the existing name or the new name.
- “QlgLstat()—Get File or Link Information (using NLS-enabled path name)” on page 265 (Get file or link information (using NLS-enabled path name)) gets status information about a specified file and places it in the area of memory pointed to by buf.
- “QlgLstat64()—Get File or Link Information (large file enabled and using NLS-enabled path name)” on page 267 (Get file or link information (large file enabled and using NLS-enabled path name)) gets status information about a specified file and places it in the area of memory pointed to by buf.
- “QlgMkdir()—Make Directory (using NLS-enabled path name)” on page 270 (Make directory (using NLS-enabled path name)) creates a new, empty directory whose name is defined by path.
- “QlgMkfifo()—Make FIFO Special File (using NLS-enabled path name)” on page 271 (Make FIFO special file (using NLS-enabled path name)) creates a new FIFO special file whose name is defined by path.

- “QlgOpen()—Open a File (using NLS-enabled path name)” on page 273 (Open a file (using NLS-enabled path name)) opens a file or creates a new, empty file whose name is defined by path and returns a number called a file descriptor.
- “QlgOpen64()—Open File (large file enabled and using NLS-enabled path name)” on page 274 (Open file (large file enabled and using NLS-enabled path name)) opens a file and returns a number called a file descriptor.
- “QlgOpendir()—Open Directory (using NLS-enabled path name)” on page 275 (Open directory (using NLS-enabled path name)) opens a directory so it can be read.
- “QlgPathconf()—Get Configurable Path Name Variables (using NLS-enabled path name)” on page 278 (Get configurable path name variables (using NLS-enabled path name)) lets an application determine the value of a configuration variable (name) associated with a particular file or directory (path).
- “QlgProcessSubtree()—Process a Path Name (using NLS-enabled path name)” on page 279 (Process a path name (using NLS-enabled path name)) searches the directory tree under a specific path name.
- “QlgReaddir()—Read Directory Entry (using NLS-enabled path name)” on page 280 (Read directory entry (using NLS-enabled path name)) returns a pointer to a structure describing the next directory entry in the directory stream associated with dirp.
- “QlgReaddir_r()—Read Directory Entry (using NLS-enabled path name)” on page 282 (Read directory entry (using NLS-enabled path name)) initializes a structure that is referenced by entry to represent the next directory entry in the directory stream that is associated with dirp.
- “QlgReadlink()—Read Value of Symbolic Link (using NLS-enabled path name)” on page 284 (Read value of symbolic link (using NLS-enabled path name)) places the contents of the symboliclink path in the buffer buf.
- “QlgRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists (using NLS-enabled path name)” on page 286 (Rename file or directory, keep ″new″ if it exists (using NLS-enabled path name)) renames a file or a directory specified by old to the name given by new.
- “QlgRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists (using NLS-enabled path name)” on page 288 (Rename file or directory, unlink ″new″ if it exists (using NLS-enabled path name)) renames a file or a directory specified by old to the name given by new.
- “QlgRmdir()—Remove Directory (using NLS-enabled path name)” on page 290 (Remove directory (using NLS-enabled path name)) removes a directory, path, provided that the directory is empty; that is, the directory contains no entries other than ′dot′ (.) or ′dot-dot′ (..).
- “QlgSaveStgFree()—Save Storage Free (using NLS-enabled path name)” on page 292 (Save Storage Free (using NLS-enabled path name)) calls a user-supplied exit program to save an *STMF iSeries object type and, upon successful completion of the exit program, frees the storage for the object and marks the object as storage freed.
- “QlgSetAttr()—Set Attributes (using NLS-enabled path name)” on page 292 (Set attributes (using NLS-enabled path name)) sets one of a set of defined attributes, on each call, for the object that is referred to by the input *Path_Name.
- “QlgStat()—Get File Information (using NLS-enabled path name)” on page 293 (Get file information (using NLS-enabled path name)) gets status information about a specified file and places it in the area of memory pointed to by the buf argument.
- “QlgStat64()—Get File Information (large file enabled and using NLS-enabled path name)” on page 295 (Get file information (large file enabled and using NLS-enabled path name)) gets status information about a specified file and places it in the area of memory pointed to by the buf argument.
- “QlgStatvfs()—Get File System Information (using NLS-enabled path name)” on page 296 (Get file system information (using NLS-enabled path name)) gets status information about the file system that contains the file named by the path argument.
- “QlgStatvfs64()—Get File System Information (64-Bit enabled and using NLS-enabled path name)” on page 298 (Get file system information (64-bit enabled and using NLS-enabled path name)) gets status information about the file system that contains the file named by the path argument.

- "QlgSymlink()—Make Symbolic Link (using NLS-enabled path name)" on page 300 (Make symbolic link (using NLS-enabled path name)) creates the symbolic link named by slink with the value specified by pname.
- "QlgUnlink()—Remove Link to File (using NLS-enabled path name)" on page 302 (Remove link to file (using NLS-enabled path name)) removes a directory entry that refers to a file.
- "QlgUtime()—Set File Access and Modification Times (using NLS-enabled path name)" on page 303 (Set file access and modification times (using NLS-enabled path name)) sets the access and modification times of path to the values in the utimbuf structure.
- "Perform Miscellaneous File System Functions (QP0FPTOS) API" on page 305 (Perform Miscellaneous File System Functions) performs a variety of file system functions.
- QP0LCHSG (Change Scan Signature) changes the scan key signature associated with a specific scan key.
- "Qp0lCvtPathToQSYSObjName()— Resolve Integrated File System Path Name into QSYS Object Name" on page 308 (Resolve integrated file system path name into QSYS object name) resolves a given integrated file system path name into the three-part QSYS.LIB file system name: library, object, and member.
- "Perform File System Operation (QP0LFLOP) API" on page 315 (Perform file system operation) performs miscellaneous file system operations.
- "Qp0lGetAttr()—Get Attributes" on page 326 (Get attributes) gets one or more attributes, on a single call, for the object that is referred to by the input Path_Name.
- "Qp0lGetPathFromFileID()—Get Path Name of Object from Its File ID" on page 351 (Get path name of object from its file ID) determines an absolute path name of the file identified by fileid and stores it in buf.
- "Qp0lOpen()—Open File" on page 354 (Open file) opens a file and returns a number called a file descriptor.
- "Qp0lProcessSubtree()—Process a Path Name" on page 356 (Process a path name) searches the directory tree under a specific path name. It selects and passes objects, one at a time, to an exit program that is identified on its call. The exit program can be either a procedure or a program.
- "Qp0lRenameKeep()—Rename File or Directory, Keep "new" If It Exists" on page 373 (Rename file or directory, keep*new*if it exists) renames a file or a directory specified by old to the name given by new.
- "Qp0lRenameUnlink()—Rename File or Directory, Unlink "new" If It Exists" on page 379 (Rename file or directory, unlink*new*if it exists) renames a file or a directory specified by old to the name given by new.
- "Retrieve Object References (QP0LROR)" on page 385 (Retrieve Object References) retrieves information about integrated file system references on an object.
- QP0LRRO (Retrieve Referenced Objects) retrieves usage information about integrated file system objects that have been referenced by a specified job.
- QP0LRTSG (Retrieve Scan Signature) retrieves the scan key signature associated with a specific scan key.
- "Qp0lSaveStgFree()—Save Storage Free" on page 399 (Save Storage Free) calls a user-supplied exit program to save an *STMF iSeries object type and, upon successful completion of the exit program, frees the storage for the object and marks the object as storage freed.
- "Qp0lSetAttr()—Set Attributes" on page 403 (Set attributes) renames a file or a directory specified by old to the name given by new.
- "Qp0lUnlink()—Remove Link to File" on page 418 (Remove link to file) removes a directory entry that refers to a file.
- "Qp0zPipe()—Create Interprocess Channel with Sockets" on page 419 (Create interprocess channel with sockets) creates a data pipe that can be used by two processes.
- "qsygetgroups()—Get Supplemental Group IDs" on page 420 (Get Supplemental Group IDs) returns the supplemental group IDs associated with the calling thread.

- "qsysetegid()—Set Effective Group ID" on page 421 (Set effective group ID) sets the effective group ID to gid.
- "qsyseteuid()—Set Effective User ID" on page 423 (Set effective user ID) sets the effective user ID to uid.
- "qsysetgid()—Set Group ID" on page 424 (Set group ID) sets the real, effective and saved groups to gid.
- "qsysetgroups()—Set Supplemental Group IDs" on page 425 (Set Supplemental Group IDs) sets the supplementary group IDs of the calling thread.
- "qsysetregid()—Set Real and Effective Group IDs" on page 427 (Set real and effective group IDs) is used to set the real and effective group IDs. The real and effective group IDs may be set to different values in the same call.
- "qsysetreuid()—Set Real and Effective User IDs" on page 428 (Set real and effective user IDs) sets the real and effective user IDs to the values specified by ruid and euid.
- "qsysetuid()—Set User ID" on page 430 (Set user ID) sets the real, effective, and saved user ID to uid.
- "Retrieve Network File System Export Entries (QZNFRTVE) API" on page 431 (Retrieve network file system export entries) returns the list of Network File System (NFS) export entries for objects currently exported to NFS clients or for objects referenced in the /etc/exports file.
- "read()—Read from Descriptor" on page 437 (Read from Descriptor) reads nbyte bytes of input into the memory area indicated by buf.
- "readdir()—Read Directory Entry" on page 443 (Read directory entry) returns a pointer to a dirent structure describing the next directory entry in the directory stream associated with dirp.
- "readdir_r()—Read Directory Entry" on page 447 (Read directory entry) initializes the dirent structure that is referenced by entry to represent the next directory entry in the directory stream that is associated with dirp.
- "readdir_r_ts64()—Read Directory Entry" on page 451 (Read directory entry) initializes the dirent structure that is referenced by entry to represent the next directory entry in the directory stream that is associated with dirp.
- "readlink()—Read Value of Symbolic Link" on page 452 (Read value of symbolic link) places the contents of the symbolic link path in the buffer buf.
- "readv()—Read from Descriptor Using Multiple Buffers" on page 455 (Read from Descriptor Using Multiple Buffers) is used to receive data from a file or socket descriptor.
- "rename()—Rename File or Directory" on page 460 (Rename file or directory) is used to rename a file or directory with the semantics of Qp0lRenameUnlink() or Qp0lRenameKeep().
- "rewinddir()—Reset Directory Stream to Beginning" on page 461 (Reset directory stream) 'rewinds' the position of an open directory stream to the beginning.
- "rmdir()—Remove Directory" on page 463 (Remove directory) removes a directory, path, provided that the directory is empty; that is, the directory contains no entries other than 'dot' (.) or 'dot-dot' (..).
- "stat()—Get File Information" on page 468 (Get file information) gets status information about a specified file and places it in the area of memory pointed to by the buf argument.
- "stat64()—Get File Information (Large File Enabled)" on page 475 (Get file information (large file enabled)) gets status information about a specified file and places it in the area of memory pointed to by the buf argument.
- "statvfs()—Get File System Information" on page 478 (Get file system information) gets status information about the file system that contains the file named by the path argument.
- "statvfs64()—Get File System Information (64-Bit Enabled)" on page 483 (Get file system information (large file enabled)) gets status information about the file system that contains the file named by the path argument.
- "symlink()—Make Symbolic Link" on page 485 (Make symbolic link) creates the symbolic link named by slink with the value specified by pname.

- “sysconf()—Get System Configuration Variables” on page 488 (Get system configuration variables) returns the value of a system configuration option.
- “umask()—Set Authorization Mask for Job” on page 491 (Set authorization mask for job) changes the value of the file creation mask for the current job to the value specified in cmask.
- “unlink()—Remove Link to File” on page 492 (Remove link to file) removes a directory entry that refers to a file.
- “utime()—Set File Access and Modification Times” on page 497 (Set file access and modification times) sets the access and modification times of path to the values in the utimbuf structure.
- “write()—Write to Descriptor” on page 502 (Write to Descriptor) writes nbyte bytes from buf to the file or socket associated with file_descriptor.
- “writev()—Write to Descriptor Using Multiple Buffers” on page 509 (Write to Descriptor Using Multiple Buffers) is used to write data to a file or socket descriptor.

The integrated file system exit programs are:

- “Integrated File System Scan on Close Exit Program” on page 513 is called during close processing such as with the “close()—Close File or Socket Descriptor” on page 34API. This exit program must be provided by the user.
- “Integrated File System Scan on Open Exit Program” on page 523 is called during open processing such as with the “open()—Open File” on page 195API. This exit program must be provided by the user.
- “Process a Path Name Exit Program” on page 533 is called by the “Qp0lProcessSubtree()—Process a Path Name” on page 356API for each object in the API’s search that meets the caller’s selection criteria. This exit program must be provided by the user.
- “Save Storage Free Exit Program” on page 535 is called by the “Qp0lSaveStgFree()—Save Storage Free” on page 399 API to save an *STMF iSeries object type.

In addition to the functions above, the following functions, which are described in the Sockets APIs, also can operate on files in the integrated file system.

Other Functions that Operate on Files

Function	Description
givedescriptor()	Give file access to another job Give socket access to another job
select()	Check I/O status of multiple file descriptors Wait for events on multiple sockets
takedescriptor()	Take file access from another job Take socket access from another job

Note: These functions use header (include) files from the library QSYSINC, which is optionally installable. Make sure QSYSINC is installed on your system before using any of the functions. See “Header Files for UNIX-Type Functions” on page 537) for the file and member name of each header file.

Many of the terms used in this chapter, such as current directory, file system, path name, and link, are explained in the Integrated file system information. The API Examples also shows an example of using several integrated file system functions.

To determine whether a particular function updates the access, change, and modification times of the object on which it performs an operation, see “Integrated File System APIs—Time Stamp Updates” on page 548.

APIs

These are the APIs for this category.

access()—Determine File Accessibility

Syntax

```
#include <unistd.h>

int access(const char *path, int amode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 12.

The **access()** function determines whether a file can be accessed in a particular manner. When checking whether a job has appropriate permissions, **access()** looks at the *real* user ID (UID) and group ID (GID), not the effective IDs. Adopted authority is not used.

Parameters

path

(Input) A pointer to the null-terminated path name for the file to be checked for accessibility.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

`const char` **path* is the name of the file whose accessibility you want to determine. If the named file is a symbolic link, **access()** resolves the symbolic link.

See "QlgAccess()—Determine File Accessibility (using NLS-enabled path name)" on page 235—Determine File Accessibility (using NLS-enabled path name) for a description and an example of supplying the *path* in any CCSID.

amode

(Input) A bitwise representation of the access permissions to be checked.

The following symbols, which are defined in the <**unistd.h**> header file, can be used in *amode*:

F_OK Tests whether the file exists

R_OK Tests whether the file can be accessed for reading

W_OK Tests whether the file can be accessed for writing

X_OK Tests whether the file can be accessed for execution

You can take the bitwise inclusive OR of any or all of the last three symbols to test several access modes at once. If you are using F_OK to test for the existence of the file, you cannot use OR with any of the other symbols. If any other bits are set in *amode*, **access()** returns the [EINVAL] error.

If the job has *ALLOBJ special authority, **access()** will indicate success for R_OK, W_OK, or X_OK even if none of the permission bits are set.

Authorities

Authorization Required for access()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be tested	*X	EACCES
Object when R_OK is specified	*R	EACCES
Object when W_OK is specified	*W	EACCES
Object when X_OK is specified	*X	EACCES
Object when R_OK ∣ W_OK is specified	*RW	EACCES
Object when R_OK ∣ X_OK is specified	*RX	EACCES
Object when W_OK ∣ X_OK is specified	*WX	EACCES
Object when R_OK ∣ W_OK ∣ X_OK is specified	*RWX	EACCES
Object when F_OK is specified	None	None

Return Value

0 **access()** was successful.

-1 **access()** was not successful (the specified access is not permitted). The *errno* global variable is set to indicate the error.

Error Conditions

If **access()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EINTR (page 541)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	

Error condition	Additional information
[ENOTSUP (page 542)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETXTBSY (page 547)]	
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
   - Where multiple threads exist in the job.
   - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB

- Independent ASP QSYS.LIB
  - QOPT
  - Network File System
  - QFileSvr.400
2. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)
3. QOPT File System Differences

   If the object exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the object and preceding directories in the path name follows the rules described in Authorization Required for access() (page 11) . If the object exists on a volume formatted in some other media format, no authorization checks are made on the object or preceding directories. The volume authorization list is checked for the requested authority regardless of the volume media format.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**limits.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "accessx()—Determine File Accessibility for a Class of Users" on page 14—Determine File Accessibility for Class of Users
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "faccessx()—Determine File Accessibility for a Class of Users" on page 61—Determine File Accessibility for Class of Users
- "open()—Open File" on page 195—Open File
- "QlgAccess()—Determine File Accessibility (using NLS-enabled path name)" on page 235Determine File Accessibility using NLS-enabled path name)
- "QlgAccessx()—Determine File Accessibility for a Class of Users (using NLS-enabled path name)" on page 237—Determine File Accessibility for Class of Users (using NLS-enabled path name)
- "stat()—Get File Information" on page 468—Get File Information

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines how a file is accessed:

```
#include <stdio.h>
#include <unistd.h>

main() {
  char path[]="/";

  if (access(path, F_OK) != 0)
    printf("'%s' does not exist!\n", path);
  else {
    if (access(path, R_OK) == 0)
      printf("You have read access to '%s'\n", path);
    if (access(path, W_OK) == 0)
```

```
      printf("You have write access to '%s'\n", path);
    if (access(path, X_OK) == 0)
      printf("You have search access to '%s'\n", path);
  }
}
```

Output:

The output from a user with read and execute access is:

```
You have read access to '/'
You have search access to '/'
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

accessx()—Determine File Accessibility for a Class of Users

Syntax

```
#include <unistd.h>

int accessx(const char *path, int amode, int who);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 17.

The **accessx()** function determines whether a file can be accessed by a specified class of users in a particular manner. The caller must have authority to all components in the path name prefix. Adopted authority is not used.

Parameters

path (Input) A pointer to the null-terminated path name for the file to be checked for accessibility.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

`const char` **path* is the name of the file whose accessibility you want to determine. If the named file is a symbolic link, **accessx()** resolves the symbolic link.

See "QlgAccessx()—Determine File Accessibility for a Class of Users (using NLS-enabled path name)" on page 237— Determine File Accessibility for Class of Users (using NLS-enabled path name) for a description and an example of supplying the *path* in any CCSID.

amode (Input) A bitwise representation of the access permissions to be checked.

The following symbols, which are defined in the <**unistd.h**> header file, can be used in *amode*:

F_OK (*x'00'*) Tests whether the file exists

R_OK (*x'04'*) Tests whether the file can be accessed for reading

W_OK (*x'02'*) Tests whether the file can be accessed for writing

X_OK (*x'01'*) Tests whether the file can be accessed for execution

You can take the bitwise inclusive OR of any or all of the last three symbols to test several access modes at once. If you are using F_OK to test for the existence of the file, you cannot use OR with any of the other symbols. If any other bits are set in *amode*, **accessx()** returns the [EINVAL] error.

who (Input) The class of users whose authority is to be checked.

The following symbols, which are defined in the <**unistd.h**> header file, can be used in *who*:

`ACC_SELF`
(*x'00'*) Determines if specified access is permitted for the current thread. The effective user and group IDs are used.

Note: If the real and effective user ID are the same and the real and effective group ID are the same, the request is treated as ACC_INVOKER. See the Usage Notes for more details.

`ACC_INVOKER`
(*x'01'*) Determines if specified access is permitted for the current thread. The real user and group IDs are used.

Note: The expression **access**(*path, amode*) is equivalent to **accessx**(*path, amode,* **ACC_INVOKER**)

`ACC_OTHERS`
(*x'08'*) Determines if specified access is permitted for any user other than the object owner. Only one of *R_OK*, *W_OK*, and *X_OK* is permitted when *who* is *ACC_OTHERS*. Privileged users (users with *ALLOBJ special authority) are not considered in this check.

`ACC_ALL`
(*x'20'*) Determines if specified access is permitted for all users. Only one of *R_OK*, *W_OK*, and *X_OK* is permitted when **who** is *ACC_ALL*. Privileged users (users with *ALLOBJ special authority) are not considered in this check.

Authorities

Authorization Required to Path Prefix for accessx()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be tested	*X	EACCES

The following authorities are required if the **who** parameter is *ACC_SELF* or *ACC_INVOKER*. If *ACC_SELF* is specified, the effective UID and GID of the caller are used. If *ACC_INVOKER* is used, the real UID and GID of the caller are used.

Authorization Required to Object for accessx()

Object Referred to	Authority Required	errno
Object when R_OK is specified	*R	EACCES
Object when W_OK is specified	*W	EACCES
Object when X_OK is specified	*X	EACCES
Object when R_OK | W_OK is specified	*RW	EACCES
Object when R_OK | X_OK is specified	*RX	EACCES
Object when W_OK | X_OK is specified	*WX	EACCES
Object when R_OK | W_OK | X_OK is specified	*RWX	EACCES
Object when F_OK is specified	None	None

If the thread has *ALLOBJ special authority, **accessx()** with *ACC_SELF* or *ACC_INVOKER* will indicate success for R_OK, W_OK, or X_OK even if none of the permission bits are set.

Return Value

0 **accessx()** was successful.

-1 **accessx()** was not successful (or the specified access is not permitted for the class of users being checked). The *errno* global variable is set to indicate the error.

Error Conditions

If **access()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	The class of users specified by the *who* parameter does not have the permission indicated by the *amode* parameter. If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EINTR (page 541)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETXTBSY (page 547)]	
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	

Error condition	Additional information
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
   - Where multiple threads exist in the job.
   - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. ACC_SELF Mapped to ACC_INVOKER

   Some physical file systems do not support *ACC_SELF* for the *who* parameter. Therefore, **accessx()** will change the *who* parameter from *ACC_SELF* to *ACC_INVOKER* if the caller's real and effective user ID are equal, and the caller's real and effective group ID are equal.
3. Network File System Differences

   The Network File System will only support the value *ACC_INVOKER* for the *who* parameter. If **accessx()** is called on a file in a mounted Network File System directory with a value for *who* other

than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

4. QNTC File System Differences

   The QNTC File System will only support the value *ACC_INVOKER* for the *who* parameter. If **accessx()** is called on a file in the QNTC File System with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

5. QOPT File System Differences

   If the object exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the object and preceding directories in the path name follows the rules described in the previous table (page 15), Authorization Required to Object for **accessx()**. If the object exists on a volume formatted in some other media format, no authorization checks are made on the object or preceding directories. The volume authorization list is checked for the requested authority regardless of the volume media format.

6. QFileSvr.400 File System Differences

   The QFileSvr.400 File System will only support the value *ACC_INVOKER* for the *who* parameter. If **accessx()** is called on a file in the QFileSvr.400 File System with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

7. QNetWare File System Differences

   The QNetWare File System will only support the value *ACC_INVOKER* for the *who* parameter. If **accessx()** is called on a file in the QNetWare File System with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**limits.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "open()—Open File" on page 195—Open File
- "access()—Determine File Accessibility" on page 10—Determine File Accessibility
- "faccessx()—Determine File Accessibility for a Class of Users" on page 61—Determine File Accessibility for a Class of Users
- "QlgAccessx()—Determine File Accessibility for a Class of Users (using NLS-enabled path name)" on page 237—Determine File Accessibility for a Class of Users (using NLS-enabled path name)
- "QlgAccess()—Determine File Accessibility (using NLS-enabled path name)" on page 235—Determine File Accessibility (using NLS-enabled path name)

- “stat()—Get File Information” on page 468—Get File Information

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines how a file is accessed:

```
#include <stdio.h>
#include <unistd.h>

main() {
  char path[]="/myfile";

  if (accessx(path, R_OK, ACC_OTHERS) == 0)
    printf("Someone besides the owner has read access to '%s'\n", path);
  if (accessx(path, W_OK, ACC_OTHERS) == 0)
    printf("Someone besides the owner has write access to '%s'\n", path);
  if (accessx(path, X_OK, ACC_OTHERS) == 0)
    printf("Someone besides the owner has search access to '%s'\n", path);
}
```

Output:

In this example **accessx()** was called on '/myfile'. The following would be the output if someone other than the owner has *R authority, someone besides the owner has *W authority, and noone other than the owner has *X authority.

```
Someone besides the owner has read access to '/'
Someone besides the owner has write access to '/'
```

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

---

chdir()—Change Current Directory

Syntax

```
#include <unistd.h>

int chdir(const char *path);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 21.

The **chdir()** function makes the directory named by *path* the new current directory. If the last component of *path* is a symbolic link, **chdir()** resolves the contents of the symbolic link. If the **chdir()** function fails, the current directory is unchanged.

Parameters

path
(Input) A pointer to the null-terminated path name of the directory that should become the current directory.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See “QlgChdir()—Change Current Directory (using NLS-enabled path name)” on page 239—Change Current Directory for a description and an example of supplying the *path* in any CCSID.

Authorities

Note: Adopted authority is not used.

Authorization Required for chdir()

Object Referred to	Authority Required	errno
Each directory of the path name	*X	EACCES

Return Value

0 **chdir()** was successful.

-1 **chdir()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **chdir()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     – ″Root″ (/)
     – QOpenSys
     – User-defined
     – QNTC
     – QSYS.LIB
     – Independent ASP QSYS.LIB
     – QOPT
     – Network File System
     – QFileSvr.400

The **chdir()** API operates on two objects: the previous current working directory and the new one. If either of these objects is managed by a file system that is not threadsafe, **chdir()** fails with the ENOTSAFE error code.

2. QOPT File System Differences

   If the directory exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for each directory in the path name follows the rules described in Authorization Required for chdir() (page 20). If the directory exists on a volume formatted in some other media format, no authorization checks are made on each directory in the path name. The volume authorization list is checked for *USE authority regardless of the volume media format.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**limits.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "fchdir()—Change Current Directory by Descriptor" on page 66—Change Current Directory by Descriptor
- "getcwd()—Get Current Directory" on page 113—Get Current Directory
- "QlgChdir()—Change Current Directory (using NLS-enabled path name)" on page 239—Change Current Directory
- "QlgGetcwd()—Get Current Directory (using NLS-enabled path name)" on page 248—Get Current Directory

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **chdir()**:

```
#include <stdio.h>
#include <unistd.h>

main() {
  if (chdir("/tmp") != 0)
    perror("chdir() to /tmp failed");
  if (chdir("/chdir/error") != 0)
    perror("chdir() to /chdir/error failed");
}
```

Output:

```
chdir() to /chdir/error failed: No such path or directory.
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

chmod()—Change File Authorizations

Syntax

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 26.

The **chmod()** function changes S_ISUID, S_ISGID, S_ISVTX, and the permission bits of the file or directory specified in *path* to the corresponding bits specified in *mode*. If the named file is a symbolic link, **chmod()** resolves the symbolic link. **chmod()** has no effect on file descriptions for files that are open at the time **chmod()** is called.

When **chmod()** is successful it updates the change time of the file.

If the file is checked out by another user (someone other than the user profile of the current job), **chmod()** fails with the [EBUSY] error.

Parameters

path
(Input) A pointer to the null-terminated path name of the file whose mode is being changed.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240 for a description and an example of supplying the *path* in any CCSID.

mode
(Input) Bits that define S_ISUID, S_ISGID, S_ISVTX, and the access permissions of the file.

» The *mode* argument can be one of the following symbols defined in the <**sys/stat.h**> include file, or constructed with a bitwise inclusive OR of two or more of these symbols. « See the "Usage Notes" on page 26 for the file system differences regarding these symbols.

S_IRUSR
Read permission for the file owner

S_IWUSR
Write permission for the file owner

S_IXUSR
Search permission (for a directory) or execute permission (for a file) for the file owner

S_IRWXU
Read, write, and search or execute for the file owner. S_IRWXU is the bitwise inclusive OR of S_IRUSR, S_IWUSR, and S_IXUSR

S_IRGRP
Read permission for the file's group

S_IWGRP
Write permission for the file's group

S_IXGRP
Search permission (for a directory) or execute permission (for a file) for the file's group

S_IRWXG
Read, write, and search or execute permission for the file's group. S_IRWXG is the bitwise inclusive OR of S_IRGRP, S_IWGRP, and S_IXGRP

S_IROTH
General read permission

S_IWOTH
General write permission

S_IXOTH
General search permission (for a directory) or general execute permission (for a file)

S_IRWXO
: General read, write, and search or execute permission. S_IRWXO is the bitwise inclusive OR of S_IROTH, S_IWOTH, and S_IXOTH

S_ISUID
: Set effective user ID at execution time. This bit is ignored if the object specified by *path* is a directory.

S_ISGID
: Set effective group ID at execution time. See "Usage Notes" on page 26 for more information if the object specified by *path* is a directory.

S_ISVTX
: Restricted renames and unlinks for objects within a directory. Objects can be linked into a directory that has this bit set on, but cannot be renamed or unlinked from it unless one or more of the following are true for the user performing the operation:

- The user is the owner of the object.
- The user is the owner of the directory.
- The user has all object (*ALLOBJ) special authority.

This restriction only applies to directories in the "root" (/), QOpenSys, and user-defined file systems. Other types of object and directories in other file systems may have this bit on, however, it will be ignored.

If bits other than the bits listed above are set in *mode*, **chmod()** returns the [EINVAL] error.

Authorities

Note: Adopted authority is not used.

Authorization required for chmod() (excluding QDLS, QSYS.LIB, and Independent ASP QSYS.LIB)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	Owner (see **Note**)	EPERM
Note: You do not need the listed authority if you have *ALLOBJ special authority.		

Authorization required for chmod() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	Owner or *ALL	EACCES

Authorization required for chmod() in the QSYS.LIB and Independent ASP QSYS.LIB file systems.

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES

Object Referred to	Authority Required	errno
The parent directory of the object if the object is a save file	*RX	EPERM
Object	Owner (see **Note**)	EPERM
Note: You do not need the listed authority if you have *ALLOBJ special authority.		

Return Value

0 **chmod()** was successful.

-1 **chmod()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **chmod()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

Error condition	Additional information
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400

2. ″Root″ (/), QOpenSys, and User-Defined File System Differences

   If the object has a primary group, it must match the primary group ID or one of the supplemental group IDs of the caller of the API; otherwise, the S_ISGID bit is turned off.

3. QSYS.LIB and independent ASP QSYS.LIB File System Differences

   chmod() is not supported for member (.MBR) objects.

   chmod() returns EBUSY if the object is allocated in another job.

   QSYS.LIB and independent ASP QSYS.LIB do not support setting the S_ISUID (set-user-ID), S_ISGID (set-group_ID), and S_ISVTX (restricted rename and unlink) bits. If they are turned on in the mode parameter, they are ignored.

4. QDLS File System Differences

   Changing the permissions of the /QDLS directory (the root folder) is not allowed. If an attempt is made to change the permissions, error ENOTSUP is returned.

   "Group" rights are not set if there is no current group.

   QDLS does not support setting the S_ISUID, S_ISGID, and S_ISVTX bits. If they are turned on in the mode parameter, they are ignored.

5. QOPT File System Differences

   Changing the permissions is allowed only for an object that exists on a volume formatted in Universal Disk Format (UDF). For all other media formats, ENOTSUP is returned.

   In addition to the authorization checks described in Authorization Required for chmod() (page 24), the volume authorization list is checked for *CHANGE authority.

   QOPT does not support setting the S_ISUID, S_ISGID, and S_ISVTX bits for any optical media format. If they are turned on in the mode parameter, ENOTSUP is returned.

6. QNetWare File System Differences

   The QNetWare file system does not fully support **chmod()**. See NetWare on iSeries for more information.

   QNetWare supports the S_ISUID and S_ISGID bits by passing them to the server and surfacing them to the caller. Some versions of NetWare may support the bits and others may not.

   QNetWare does not support setting the S_ISVTX bit. If it is turned on in the mode parameter, ENOTSUP is returned.

7. QFileSvr.400 Differences

   QFileSvr.400 supports the S_ISUID, S_ISGID, and S_ISVTX bits by passing them to the server and surfacing them to the caller.

8. Network File System Differences

   The NFS client supports the S_ISUID, S_ISGID, and S_ISVTX bits by passing them to the server over the network and surfacing them to the caller. Whether a particular network file system supports the setting of these bits depends on the server. Most servers have the capability of masking off the S_ISUID and S_ISGID bits if the NOSUID option is specified on the export. The default, however, is to support these two bits.

9. QNTC File System Differences

   chmod() does not update the Windows NT server access control lists that control the authority of users to the file or directory. The mode settings are ignored.

10. S_ISGID bit of a directory in ″root″ (/), QOpenSys, or User-Defined File System

    The S_ISGID bit of the directory affects what the group ID (GID) is for objects that are created in the directory. If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the S_ISGID bit of the parent directory is on, the group ID (GID) of the new object is set to the GID of the parent directory. For all other file systems, the GID of the new object is set to the GID of the parent directory.

Related Information

- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**sys/stat.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chown()—Change Owner and Group of File" on page 29—Change Owner and Group of File
- "fchmod()—Change File Authorizations by Descriptor" on page 69—Change File Authorizations by Descriptor
- "mkdir()—Make Directory" on page 169—Make Directory
- "open()—Open File" on page 195—Open File
- "stat()—Get File Information" on page 468—Get File Information
- "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240—Change File Authorizations

Example

See Code disclaimer information for information pertaining to code examples.

The following example changes the permissions for a file:

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

main() {
  char fn[]="temp.file";
  int file_descriptor;
  struct stat info;

  if ((file_descriptor = creat(fn, S_IWUSR)) == -1)
    perror("creat() error");
  else {
    if (stat(fn, &info)!= 0)
       perror("stat() error");
    else {
      printf("original permissions were: %08o\n", info.st_mode);
    }
    if (chmod(fn, S_IRWXU|S_IRWXG) != 0)
      perror("chmod() error");
    else {
      if (stat(fn, &info)!= 0)
         perror("stat() error");
      else {
         printf("after chmod(), permissions are: %08o\n", info.st_mode);
      }
    }
    if (close(file_descriptor)!= 0)
       perror("close() error");
    if (unlink(fn)!= 0)
       perror("unlink() error");
  }
}
```

Output:

```
original permissions were: 00100200
after chmod(), permissions are: 00100770
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

chown()—Change Owner and Group of File

Syntax

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 32.

The **chown()** function changes the owner and primary group of a file. If the named file is a symbolic link, **chown()** resolves the symbolic link. The permissions of the previous owner or primary group to the object are revoked.

If the file is checked out by another user (someone other than the user profile of the current job), **chown()** fails with the [EBUSY] error.

When **chown()** completes successfully, it updates the change time of the file.

Parameters

path

(Input) A pointer to the null-terminated path name of the file whose owner and group are being changed.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See “QlgChown()—Change Owner and Group of File (using NLS-enabled path name)” on page 242 for a description and an example of supplying the *path* in any CCSID.

owner

(Input) The user ID (UID) of the new owner of the file.

group

(Input) The group ID (GID) of the new primary group for the file.

» The new primary group user cannot be the owner of the object. «

Note: Changing the owner or the primary group causes the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode to be cleared, unless the caller has all object (*ALLOBJ) special authority. If the caller does have *ALLOBJ special authority, the bits are not changed. This does not apply to directories or FIFO special files. See the “chmod()—Change File Authorizations” on page 22 documentation.

Authorities

Note: Adopted authority is not used.

Authorization Required for chown() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES

Object Referred to	Authority Required	errno
Object, when changing the owner	Owner and *OBJEXIST (also see **Note 1**)	EPERM
Object, when changing the primary group	See **Note 2**	EPERM
Previous owner's user profile, when changing the owner	*DLT	EPERM
New owner's user profile, when changing the owner	*ADD	EPERM
User profile of previous primary group, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM

Note:

1. You do not need the listed authority if you have *ALLOBJ special authority.
2. At least one of the following must be true:
   a. You have *ALLOBJ special authority.
   b. You are the owner and either of the following:
      - The new primary group is the primary group of the job.
      - The new primary group is one of the supplementary groups of the job.

Authorization Required for chown() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X See **Note 1**	EACCES
Object when changing the owner	See **Note 2**(a)	EPERM
Object when changing the primary group	See **Note 2**(b)	EPERM

Note:

1. For *FILE objects (such as DDM file, diskette file, print file, and save file), *RX authority is required to the parent directory of the object, rather than just *X authority.
2. The required authorization varies for each object type. For details of the following commands, see the iSeries Security Reference book.
   a. CHGOWN
   b. CHGPGP

Authorization Required for chown() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	*ALLOBJ Special Authority or Owner	EPERM
Previous owner's user profile, when changing the owner	*DLT	EPERM
New owner's user profile, when changing the owner	*ADD	EPERM
Previous primary group's user profile, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM

Authorization Required for chown() in the QOPT File System

Object Referred to	Authority Required	errno
Volume authorization list	*CHANGE	EACCES
Each directory in the path name preceding the object.	*X	EACCES
Object	*ALLOBJ Special Authority or Owner	EPERM

Return Value

0 **chown()** was successful.

-1 **chown()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **chown()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	The *owner* or *group* is not a valid user ID (UID) or group ID (GID). The *owner* is the current primary group of the object.
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	

Error condition	Additional information
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

Error Messages

The following messages may be sent from this API:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   chown() is not supported for member (.MBR) objects.
3. QDLS File System Differences

   The owner and primary group of the /QDLS directory (root folder) cannot be changed. If an attempt is made to change the owner and primary group, error ENOTSUP error is returned.
4. QOPT File System Differences

   Changing the owner and primary group is allowed only for an object that exists on a volume formatted in Universal Disk Format (UDF). For all other media formats, ENOTSUP will be returned.

   QOPT file system objects that have owners will not be recognized by the Work with Objects by Owner (WRKOBJOWN) CL command. Likewise, QOPT objects that have a primary group will not be recognized by the Work Objects by Primary Group) CL command.
5. QFileSvr.400 File System Differences

   The QFileSvr.400 file system does not support **chown()**.
6. QNetWare File System Differences

   The QNetWare file system does not support primary group. The GID must be zero.

7. QNTC File System Differences

   The owner of files and directories cannot be changed. All files and directories in QNTC are owned by the QDFTOWN user profile.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**limits.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "fchown()—Change Owner and Group of File by Descriptor" on page 72—Change Owner and Group of File by Descriptor
- "fstat()—Get File Information by Descriptor" on page 95—Get File Information by Descriptor
- "lstat()—Get File or Link Information" on page 162—Get File or Link Information
- "stat()—Get File Information" on page 468—Get File Information
- "QlgChown()—Change Owner and Group of File (using NLS-enabled path name)" on page 242—Change Owner and Group of File

Example

See Code disclaimer information for information pertaining to code examples.

The following example changes the owner and group of a file:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

main() {
  char fn[]="temp.file";
  int file_descriptor;
  struct stat info;

  if ((file_descriptor = creat(fn, S_IRWXU)) == -1)
    perror("creat() error");
  else {
    close(file_descriptor);
    stat(fn, &info);
    printf("original owner was %d and group was %d\n", info.st_uid,
           info.st_gid);
    if (chown(fn, 152, 0) != 0)
      perror("chown() error");
    else {
      stat(fn, &info);
      printf("after chown(), owner is %d and group is %d\n",
             info.st_uid, info.st_gid);
    }
    unlink(fn);
  }
}
```

Output:

```
original owner was 137 and group was 0
after chown(), owner is 152 and group is 0
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

close()—Close File or Socket Descriptor

Syntax

```
#include <unistd.h>

int close(int fildes);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 36.

The **close()** function closes a descriptor, *fildes*. This frees the descriptor to be returned by future **open()** calls and other calls that create descriptors.

When the last open descriptor for a file is closed, the file itself is closed. If the link count of the file is zero at that time, the space occupied by the file is freed and the file becomes inaccessible.

close() unlocks (removes) all outstanding byte locks that a job has on the associated file.

When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO is discarded and internal storage used is returned to the system.

When *fildes* refers to a socket, **close()** closes the socket identified by the descriptor.

For information about the exit point that can be associated with **close()**, see "Integrated File System Scan on Close Exit Program" on page 513.

Parameters

fildes (Input) The descriptor to be closed.

Authorities

No authorization is required. Authorization is verified during **open()**, **creat()**, or **socket()**.

Return Value

0 **close()** was successful.
-1 **close()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **close()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	

Error condition	Additional information
[EDEADLK (page 543)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOBUFS (page 542)]	
[ENOSPC (page 541)]	
[ENOSYS (page 544)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ESCANFAILURE (page 547)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

Additionally, if interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [EBADF] when *fildes* is a scan descriptor that was passed to one of the scan-related exit programs. See “Integrated File System Scan on Open Exit Program” on page 523 and “Integrated File System Scan on Close Exit Program” on page 513 for more information.
2. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
3. When a socket descriptor is closed, the system tries to send any queued data associated with the socket.

   - For `AF_INET` sockets, depending on whether the `SO_LINGER` socket option is set, queued data may be discarded.

     Note: For these sockets, the default value for the `SO_LINGER` socket option has the option flag set off (the system attempts to send any queued data with an infinite wait time).
4. A socket descriptor being shared among multiple processes is not closed until the process that issued the *close()* is the last process with access to the socket.

Related Information

- The <**unistd.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “creat()—Create or Rewrite File” on page 40—Create or Rewrite File
- “dup()—Duplicate Open File Descriptor” on page 55—Duplicate Open File Descriptor
- “dup2()—Duplicate Open File Descriptor to Another Descriptor” on page 58—Duplicate Open File Descriptor to Another Descriptor
- “fcntl()—Perform File Control Command” on page 82—Perform File Control Command
- “Integrated File System Scan on Close Exit Program” on page 513
- “open()—Open File” on page 195—Open File
- setsockopt()—Set Socket Options
- “unlink()—Remove Link to File” on page 492—Remove Link to File

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **close()**

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
```

```
main() {
  int fd1, fd2;
  char out[20]="Test string",
       fn[]="test.file",
       in[20];
  short write_error;

  memset(in, 0x00, sizeof(in));

  write_error = 0;

  if ( (fd1 = creat(fn,S_IRWXU)) == -1)
    perror("creat() error");
  else if ( (fd2 = open(fn,O_RDWR)) == -1)
    perror("open() error");
  else {
    if (write(fd1, out, strlen(out)+1) == -1) {
      perror("write() error");
      write_error = 1;
    }
    close(fd1);
    if (!write_error) {
      if (read(fd2, in, sizeof(in)) == -1)
        perror("read() error");
      else printf("string read from file was: '%s'\n", in);
    }
    close(fd2);
  }
}
```

Output:

```
string read from file was: 'Test string'
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

closedir()—Close Directory

Syntax

```
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dirp);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 39.

The **closedir()** function closes the directory stream indicated by *dirp*. It frees the buffer that **readdir()** uses when reading the directory stream.

A file descriptor is used for type `DIR`; **closedir()** closes the file descriptor.

Parameters

dirp (Input) A pointer to a `DIR` that refers to the open directory stream to be closed. This pointer is returned by the **opendir()** function.

Authorities

No authorization is required. Authorization is verified during **opendir()**.

Return Value

0	**closedir()** was successful.
-1	**closedir()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **closedir()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOSPC (page 541)]	
[ENOSYS (page 544)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	

Error condition	Additional information
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. If the *dirp* argument passed to **closedir()** does not refer to an open directory, **closedir()** returns the [EBADF] or [EFAULT] error.
3. After a call to **closedir()** the *dirp* will not point to a valid directory.

Related Information

- The <**sys/types.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**dirent.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “opendir()—Open Directory” on page 212—Open Directory
- “readdir()—Read Directory Entry” on page 443—Read Directory Entry
- “rewinddir()—Reset Directory Stream to Beginning” on page 461—Reset Directory Stream to Beginning

Example

See Code disclaimer information for information pertaining to code examples.

The following example closes a directory:

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

main() {
  DIR *dir;
  struct dirent *entry;
  int count;

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
  else {
    count = 0;
    while ((entry = readdir(dir)) != NULL) {
      printf("directory entry %03d: %s\n", ++count, entry->d_name);
    }
    closedir(dir);
  }
}
```

Output:

```
directory entry 001: .
directory entry 002: ..
directory entry 003: QSYS.LIB
directory entry 004: QDLS
directory entry 005: QOpenSys
directory entry 006: home
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

creat()—Create or Rewrite File

Syntax

```
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes.

The **creat()** function creates a new file or rewrites an existing file so that it is truncated to zero length. The function call

```
creat(path,mode);
```

is equivalent to the call

```
open(path, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

This means that the file named by *path* is created if it does not already exist, opened for writing only, and truncated to zero length. For further information, see “open()—Open File” on page 195—Open File.

The *mode* argument specifies file permission bits to be used in creating the file. For more information on *mode*, see “chmod()—Change File Authorizations” on page 22—Change File Authorizations.

Parameters

path (Input) A pointer to the null-terminated path name of the file to be created or rewritten.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

When a new file is created, the new file name is assumed to be represented in the language and country or region currently in effect for the job.

See “QlgCreat()—Create or Rewrite File (using NLS-enabled path name)” on page 244 for a description and an example of supplying the *path* in any CCSID.

mode (Input) The file permission bits to be used when creating the file. The S_ISUID (set-user-ID), S_ISGID (set-group-ID), and S_ISVTX, bits also may be specified when creating the file.

See “chmod()—Change File Authorizations” on page 22 for details on the values that can be specified for *mode*.

Authorities

Note: Adopted authority is not used.

Authorization Required for creat() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be created	*X	EACCES
Existing object	*W	EACCES
Parent directory of object to be created when object does not exist	*WX	EACCES

Authorization Required for creat() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be created	*X	EACCES
Existing object	*W	EACCES
Existing object when object is a save file	*RWX	EACCES
Parent directory of object to be created when object does not exist	*OBJMGT or *OBJALTER	EACCES
Parent directory of object to be created when object does not exist and object type is *USRSPC or save file	*RX and *Add	EACCES
Parent directory of the parent directory of object to be created when object does not exist and object being created is a physical file member	*Add	EACCES

Authorization Required for creat() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be created	*X	EACCES
Existing object	*W	EACCES
Parent directory of object to be created when object does not exist	*CHANGE	EACCES

Return Value

value **creat()** was successful. The value returned is the file descriptor for the open file.

-1 **creat()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **creat()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	The open sharing mode may conflict with another open of this file, or O_WRONLY or O_RDWR is specified and the file is checked out by another user.
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EEXIST (page 543)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	For example, unused bits in *mode* are set and should be cleared.
[EIO (page 540)]	
[EISDIR (page 544)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[EMFILE (page 543)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENFILE (page 543)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOSYSRSC (page 545)]	
[ENOTAVAIL (page 547)]	

Error condition	Additional information
[ENOTDIR (page 541)]	
[ENOTSUP (page 542)]	
[EOVERFLOW (page 546)]	The specified file exists and its size is too large to be represented in a variable of type off_t (the file is larger than 2GB minus 1 byte).
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys

- User-defined
- QNTC
- QSYS.LIB
- Independent ASP QSYS.LIB
- QOPT
- Network File System
- QFileSvr.400

2. "Root" (/), QOpenSys, and User-Defined File System Differences

   The user who creates the file becomes its owner. If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the S_ISGID bit of the parent directory is on, the group ID (GID) is copied from the parent directory in which the file is created.

   The owner, primary group, and public object authorities (*OBJEXIST, *OBJMGT, *OBJALTER, and *OBJREF) are copied from the parent directory's owner, primary group, and public object authorities. This occurs even when the new file has a different owner than the parent directory. The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except for those permissions that are also set in the file mode creation mask). The new file does not have any private authorities or authorization list. It only has authorities for the owner, primary group, and public.

3. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   When creating a member, the ownership, group profile, and authorities are all derived from the member's parent physical file. The input *mode* value is ignored.

   The group ID is obtained from the primary user profile, if a group profile exists.

   The owner object authorities are set to *OBJEXIST, *OBJMGT, *OBJALTER, and *OBJREF. The primary group object authorities are specified by options in the user profile of the job. None of the public object authorities are set.

   The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except for those permissions that are also set in the file mode creation mask). The data authorities must match the data authorities of the file in which the member is being created.

   The primary group authorities are not saved if the primary group does not exist. When a primary group is attached to the object, the object gets the authorities specified in *mode*.

   A member cannot be created in a mixed-CCSID file.

   The file access time for a database member is updated using the normal rules that apply to database files. At most, the access time is updated once per day.

4. QDLS File System Differences

   The user who creates the file becomes its owner. The group ID is copied from the parent folder in which the file is created.

   The owner object authority is set to *OBJMGT + *OBJEXIST + *OBJALTER + *OBJREF.

   The primary group and public object authority and all other authorities are copied from the parent folder.

   The owner, primary group, and public data authority (including *OBJOPR) are derived from the permissions specified in *mode* (except those permissions that are also set in the file mode creation mask).

   The primary group authorities specified in *mode* are not saved if no primary group exists.

5. QOPT File System Differences

   When the volume on which the file is being created is formatted in Universal Disk Format (UDF):

   - The authorization that is checked for the object and preceding directories in the path name follows the rules described in "Authorization Required for creat()." (page 41)

- The volume authorization list is checked for *CHANGE authority.
- The user who creates the file becomes its owner.
- The group ID is copied from the parent directory in which the file is created.
- The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except those permissions that are also set in the file mode creation mask).
- The resulting share mode is O_SHARE_NONE; therefore, the function call

  ```
  creat(path,mode);
  ```

  is equivalent to the call

  ```
  open(path,
       O_CREAT|O_WRONLY|O_TRUNC|O_SHARE_NONE,
       mode);
  ```
- The same uppercase and lowercase forms in which the names are entered are preserved. No distinction is made between uppercase and lower case when searching for names.

When the volume on which the file is being created is not formatted in Universal Disk Format (UDF):

- No authorization checks are made on the object or preceding directories in the path name.
- The volume authorization list is checked for *CHANGE authority.
- QDFTOWN becomes the owner of the file.
- No group ID is assigned to the file.
- The permissions specified in the mode are ignored. The owner, primary group, and public data authorities are set to RWX.
- For newly created files, names are created in uppercase. No distinction is made between uppercase and lowercase when searching for names.

A file cannot be created as a direct child of /QOPT.

The change and modification times of the parent directory are not updated.

6. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. The creation of a file may fail if permissions and other attributes that are stored locally by the Network File System are more restrictive than those at the server. A later attempt to create a file can succeed when the locally stored data has been refreshed. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) The creation can also succeed after the file system has been remounted.

   If you try to re-create a file that was recently deleted, the request may fail because data that was stored locally by the Network File System still has a record of the file's existence. The creation succeeds when the locally stored data has been updated.

   Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations.
7. QNetWare File System Differences

   The user who creates the file becomes the owner. Mode bits are not fully supported. See NetWare on iSeries for more information.
8. This function will fail with the [EOVERFLOW] error if the specified file exists and its size is too large to be represented in a variable of type off_t (the file is larger than 2GB minus 1 byte).
9. When you develop in C-based languages and this function is compiled with the _LARGE_FILES macro defined, it will be mapped to **creat64()**.

Related Information

- The <**fcntl.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "creat64()—Create or Rewrite a File (Large File Enabled)"—Create or Rewrite a File (Large File Enabled)
- "open()—Open File" on page 195—Open File
- "QlgCreat()—Create or Rewrite File (using NLS-enabled path name)" on page 244—Create or Rewrite File

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a file:

```
#include <stdio.h>
#include <fcntl.h>

main() {
  char fn[]="creat.file", text[]="This is a test";
  int fd, rc;

  if ((fd = creat(fn, S_IRUSR | S_IWUSR)) < 0)
    perror("creat() error");
  else {
    if (-1==(rc=write(fd, text, strlen(text))))
      perror("write() error");
    if (close(fd) != 0)
      perror("close() error");
    if (unlink(fn)!= 0)
      perror("unlink() error");
  }
}
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

creat64()—Create or Rewrite a File (Large File Enabled)

Syntax

```
#include <fcntl.h>

int creat64(const char *path, mode_t mode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 47.

The **creat64()** function creates a new file or rewrites an existing file so that it is truncated to zero length. The open file instance created with **creat64()** is allowed to be larger than 2GB minus 1 byte. The function call

```
creat64(path,mode);
```

is equivalent to the call

```
open64(path, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

If the file named by *path* does not already exist, it is created. The file is then opened for writing only and truncated to zero length. For further information, see "open64()—Open File (Large File Enabled)" on page 211—Open File (Large File Enabled).

See "QlgCreat64()—Create or Rewrite a File (large file enabled and using NLS-enabled path name)" on page 245—Create or Rewrite a File (Large File Enabled) for a description and an example of supplying the *path* in any CCSID.

The *mode* argument specifies file permission bits to be used in creating the file. For more information on *mode*, see "fchmod()—Change File Authorizations by Descriptor" on page 69—Change File Authorizations.

For additional information about parameters, authorities required, error conditions, and examples, see "creat()—Create or Rewrite File" on page 40—Create or Rewrite File.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **creat64()** API, you must compile the source with _LARGE_FILE_API macro defined.
2. All of the usage notes for **creat()** apply to **creat64()**. See "Usage Notes" on page 43 in the **creat()** API.

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

DosSetFileLocks()—Lock and Unlock a Byte Range of an Open File

Syntax

```
#define INCL_DOSERRORS
#define INCL_DOSFILEMGR
#include <os2.h>

 APIRET APIENTRY DosSetFileLocks(HFILE FileHandle,
                  PFILELOCK ppUnLockRange,
                  PFILELOCK ppLockRange,
                  ULONG ulTimeOut,
                  ULONG ulFlags);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 49.

The **DosSetFileLocks()** function locks and unlocks a range of an open file. A non-zero range indicates that a lock or unlock request is being made.

Parameters

FileHandle
(Input) The file descriptor of the file in which a range is to be locked.

ppUnlockRange
(Input) Address of the structure containing the offset and length of a range to be unlocked. The structure is as follows:

FileOffset
The offset to the beginning of the range to be unlocked.

RangeLength
The length of the range to be unlocked. A value of zero means that unlocking is not required.

ppLockRange
(Input) Address of the structure containing the offset and length of a range to be locked. The structure is as follows:

FileOffset
The offset to the beginning of the range to be locked.

RangeLength
The length of the range to be locked. A value of zero means that locking is not required.

ulTimeOut
(Input) The maximum time, in milliseconds, that the process is to wait for the requested locks.

ulFlags
(Input) Flags that describe the action to be taken. If any flags other than those listed below are specified, the error ERROR_INVALID_PARAMETER will be returned.

The following values are to be specified in *ulFlags*:

'0x0002' or QP0L_DOSSETFILELOCKS_ATOMIC
Atomic. This bit defines a request for atomic locking. If this bit is set to 1 and the lock range is equal to the unlock range, an atomic lock occurs. If this bit is set to 1 and the lock range is not equal to the unlock range, ERROR_LOCK_VIOLATION is returned.

'0x0001' or QP0L_DOSSETFILELOCKS_SHARE
Share. This bit defines the type of access that other processes may have to the file range that is being locked.

If this bit is set to 0 (the default), other processes have no access to the locked file range. The current process has exclusive access to the locked file range, which must not overlap any other locked file range.

If this bit is set to 1, the current process and other processes have shared access to the locked file range. A file range with shared access may overlap any other file range with shared access, but must not overlap any other file range with exclusive access.

Authorities

No authorization is required.

Return Value

NO_ERROR (0)
DosSetFileLocks() was successful.

value (non-zero)
DosSetFileLocks() was not successful. The *value* that is returned indicates the error.

Error Conditions

If **DosSetFileLocks()** is not successful, the value that is returned is one of the following errors. The <**bseerr.h**> header file defines these values.

[ERROR_GEN_FAILURE]

A general failure occurred.

This may result from damage in the system. Refer to messages in the job log for other possible causes.

[ERROR_INVALID_HANDLE]

An invalid file handle was found.

The file handle passed to this function is not valid.

[ERROR_LOCK_VIOLATION]

A lock violation was found.

The requested lock and unlock ranges are both zero.

[ERROR_INVALID_PARAMETER]

An invalid parameter was found.

A parameter passed to this function is not valid.

The byte range specified by the offset and length in the ppUnlockRange or ppLockRange parameters extends beyond 2GB minus 1 byte.

[ERROR_ATOMIC_LOCK_NOT_SUPPORTED]

The atomic lock operation is not supported.

The file system does not support atomic lock operations.

[ERROR_TIMER_NOT_SUPPORTED]

The lock timer value is not supported.

The file system does not support the lock timer value.

Error Messages

The system may send the following messages from this function.

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. If this function is called by a thread executing one of the scan-related exit programs (or any of its created threads), it will fail with error code [ENOTSUP]. See “Integrated File System Scan on Open Exit Program” on page 523 and “Integrated File System Scan on Close Exit Program” on page 513 for more information.
2. This function will fail with error code [ERROR_GEN_FAILURE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined

- QNTC
- QSYS.LIB
- Independent ASP QSYS.LIB
- QOPT
- QFileSvr.400

3. The following file systems do not support timer values other than 0. An attempt to a value other than 0 for the timer value results in an ERROR_TIMER_NOT_SUPPORTED error.

   Also, the following file systems do not support the atomic locking flag. If you turn on the atomic locking flag, an ERROR_ATOMIC_LOCKS_NOT_SUPPORTED error is returned.
   - "Root" (/)
   - QOpenSys
   - User-Defined File System
   - QDLS
   - QOPT
   - QNetWare
4. The following file systems do not support byte range locks. An attempt to use this API results in an ERROR_GEN_FAILURE error.
   - QSYS.LIB
   - Independent ASP QSYS.LIB
   - Network File System
   - QFileSvr.400
5. When you develop in C-based languages and this function is compiled with the _LARGE_FILES macro defined, it will be mapped to **DosSetFileLocks64()**. Additionally, the `PFILELOCK` data type will be mapped to a type `PFILELOCK64`.
6. Locks placed on character special files result in advisory locks. For more information on advisory locking, please see the "fcntl()—Perform File Control Command" on page 82.

Related Information

- The <**fcntl.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**os2.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**os2def.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**bse.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**bsedos.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**bseerr.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "DosSetFileLocks64()—Lock and Unlock a Byte Range of an Open File (Large File Enabled)" on page 51—Lock and Unlock a Byte Range of an Open File (Large File Enabled)

Example

See Code disclaimer information for information pertaining to code examples.

The following example opens, locks, and unlocks a file.

```
#define INCL_DOSERRORS
#define INCL_DOSFILEMGR
#include <os2.h>
#include <stdio.h>

#define NULL_RANGE 0L
#define LOCK_FLAGS 0

main() {
```

```
  char fn[]="lock.file";
  char buf[] =
     "Test data for locking and unlocking range of a file";
  int fd;
  ULONG lockTimeout = 2000; /* lock timeout of 2 seconds */
  FILELOCK Area;      /* area of file to lock/unlock */

  Area.Offset = 4;    /* start locking at byte 4 */
  Area.Range = 10;    /* lock 10 bytes for the file */

  /* Create a file for this example */
  fd = creat(fn, S_IWUSR | S_IRUSR);
  /* Write some data to the file */
  write(fd, buf, sizof(buf) -1);
  close(fd);

  /* Open the file */
  if ((fd = open(fn, O_RDWR) < 0)
  {
    perror("open() error");
    return;
  }

  /* Lock a range */
  rc = DosSetFileLocks((HFILE)fd,
                       NULL_RANGE,
                       &Area,
                       &LockTimeout,
                       LOCK_FLAGS);
  if(rc != 0) /* Lock failed */
  {
    perror("DosSetFileLocks() error");
  }


  /* Unlock a range */
  rc = DosSetFileLocks((HFILE)fd,
                       &Area,
                       NULL_RANGE,
                       &LockTimeout,
                       LOCK_FLAGS);
  if(rc != 0) /* Unlock failed */
  {
    perror("DosSetFileLocks() error");
  }

  close(fd);
  unlink(fn);
}
```

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

DosSetFileLocks64()—Lock and Unlock a Byte Range of an Open File (Large File Enabled)

Syntax

```
#define INCL_DOSERRORS
#define INCL_DOSFILEMGR
#include <os2.h>

APIRET APIENTRY DosSetFileLocks64(HFILE FileHandle,
```

```
PFILELOCK64 ppUnLockRange,
PFILELOCK64 ppLockRange,
ULONG ulTimeOut,
ULONG ulFlags);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes.”

The **DosSetFileLocks64()** function locks and unlocks a range of an open file. A non-zero range indicates that a lock or unlock request is being made.

The DosSetFileLocks64() treats the values specified in the PFILELOCK64 structure as unsigned.

The maximum offset that can be specified using DosSetFileLocks64() is the largest value that can be held in an 8-byte, unsigned integer, 2^{64} - 1.

The maximum length that can be specified using DosSetFileLocks64() is the largest value that can be held in an 8-byte, unsigned integer, 2^{64} - 1.

DosSetFileLocks64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:
- Using the **open64()** function (see “open64()—Open File (Large File Enabled)” on page 211).
- Using the **open()** function (see “open()—Open File” on page 195) with the O_LARGEFILE flag set in the oflag parameter. Note that the PFILELOCK64 type will hold offsets greater than 2 GB minus 1 byte.

For details about parameters, authorities required, error conditions, and examples, see “DosSetFileLocks()—Lock and Unlock a Byte Range of an Open File” on page 47.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **DosSetFileLocks64()** API and the PFILELOCK64 data type, you must compile the source with `_LARGE_FILE_API` defined.
2. For additional usage notes about this API, see “Usage Notes” on page 49 in the **DosSetFileLocks()** API.

Related Information

- The <**fcntl.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**os2.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**os2def.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**bse.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**bsedos.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**bseerr.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “DosSetFileLocks()—Lock and Unlock a Byte Range of an Open File” on page 47—Lock and Unlock a Byte Range of an Open File

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

DosSetRelMaxFH()—Change Maximum Number of File Descriptors

Syntax

```
#define INCL_DOSERRORS
#define INCL_DOSFILEMGR
#include <os2.h>

APIRET APIENTRY  DosSetRelMaxFH(PLONG pcbReqCount,
                               PULONG pcbCurMaxFH);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Yes

The **DosSetRelMaxFH()** function requests that the system change the maximum number of file descriptors for the calling process (job). The system preserves all file descriptors that are currently open.

A request to increase the maximum number of file descriptors by more than the system can accommodate will succeed. The resulting maximum will be the largest number possible, but will be less than what you requested.

A request to decrease the maximum number of file descriptors will succeed. The resulting maximum will be the smallest number possible, but may be more than what you expected. For example, assume that the current maximum is 200 and there are 150 open files. A request to decrease the maximum by 75 results in the maximum being decreased by only 50, to 150, to preserve the open file descriptors.

A request to decrease the maximum number of file descriptors to below 20 will succeed, but the maximum will never be decreased below 20.

To retrieve the current maximum number of file descriptors, without any side effects, the value pointed to by *pcbReqCount* should be set to zero.

Parameters

pcbReqCount
: (Input) A pointer to the number to be added to the maximum number of file descriptors for the calling process. If the value pointed to by *pcbReqCount* is positive, the system increases the maximum number of file descriptors. If the value pointed to by *pcbReqCount* is negative, the system decreases the maximum number of file descriptors.

pcbCurMaxFH
: (Output) A pointer to the location to receive the new total number of allocated file descriptors.

Authorities

No authorization is required.

Return Value

NO_ERROR (0)
: **DosSetRelMaxFH()** was successful. The function returns *NO_ERROR (0)* even if the system disregards or partially fulfills a request for an increase or a decrease (for example, decreasing by a smaller number than requested). You should examine the value pointed to by *pcbCurMaxFH* to determine the result of this function.

value (non-zero)
: **DosSetRelMaxFH()** was not successful. The *value* that is returned indicates the error.

Error Conditions

If **DosSetRelMaxFH()** is not successful, the value that is returned is one of the following errors. The <**bseerr.h**> header file defines these values.

[ERROR_GEN_FAILURE]

A general failure occurred.

This may result from damage in the system. Refer to messages in the job log for other possible causes.

[ERROR_PROTECTION_VIOLATION]

A protection violation occurred.

A pointer passed to this function is not a valid pointer.

Error Messages

The system may send the following messages from this function.

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. If you are using the **select()** API, you should be aware of the value of the FD_SETSIZE macro defined in the <**sys/types.h**> header file. This value is defined to be 200. This means that the fd_set structure is defined to contain 200 bits, one for each file descriptor.

   If your application uses **DosSetRelMaxFH()** to increase the maximum number of file descriptors beyond 200, you should consider defining your own value for the FD_SETSIZE macro prior to including <**sys/types.h**>. This is to ensure that the fd_set structure is defined with the correct number of bits to accommodate the actual maximum number of file descriptors.
2. The maximum number of file descriptors for this process may be obtained by using the "sysconf()—Get System Configuration Variables" on page 488 API with the _SC_OPEN_MAX parameter.
3. If this function is called by a thread executing one of the scan-related exit programs (or any of its created threads), it will fail with error code [ERROR_GEN_FAILURE]. See "Integrated File System Scan on Open Exit Program" on page 523 and "Integrated File System Scan on Close Exit Program" on page 513 for more information.

Related Information

- The <**os2.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**os2def.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**bse.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**bsedos.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**bseerr.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- **select()**—Wait for Events on Multiple Sockets
- "sysconf()—Get System Configuration Variables" on page 488—Get System Configuration Variables

Example

See Code disclaimer information for information pertaining to code examples.

The following example increases the maximum number of file descriptors by two.

```
#define INCL_DOSERRORS
#define INCL_DOSFILEMGR
#include <os2.h>
#include <stdio.h>

void main()
{
 long  ReqCount = 0; /* Number to add to maximum */
                     /* file descriptor count.  */
 ulong CurMaxFH;     /* New count of file descriptors. */
 int   rc;           /* Return code. */

 /* Find out what the initial maximum is.*/
 if ( NO_ERROR == (rc = DosSetRelMaxFH(&ReqCount, &CurMaxFH))
 {
  printf("Initial maximum = %d",CurMaxFH);

  ReqCount = 2; /* Set up to increase by 2. */

  if (NO_ERROR == (rc = DosSetRelMaxFH(&ReqCount, &CurMaxFH))
  {
   printf("    New maximum = %d",CurMaxFH);
  }
 }
 if (NO_ERROR != rc)
 {
  printf("Error = &d",rc);
 }
}
```

Output:

```
Initial maximum = 200    New maximum = 202
```

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

dup()—Duplicate Open File Descriptor

Syntax

```
#include <unistd.h>

int dup(int fildes);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Yes

The **dup()** function returns a new open file descriptor. The new descriptor refers to the same open file as *fildes* and shares any locks.

If the original file descriptor was opened in text mode, data conversion is also done on the duplicated file descriptor.

The FD_CLOEXEC flag that is associated with the new file descriptor is cleared. Refer to "fcntl()—Perform File Control Command" on page 82—Perform File Control Command for additional information about the FD_CLOEXEC flag.

Parameters

fildes (Input) A descriptor to be duplicated.

The following operations are equivalent:

```
fd = dup(fildes);
fd = fcntl(fildes,F_DUPFD,0);
```

For further information, see "fcntl()—Perform File Control Command" on page 82.

Authorities

No authorization is required.

Return Value

value	**dup()** was successful. The value returned is the new descriptor.
-1	**dup()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **dup()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[ECANCEL (page 543)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ENOSYS (page 544)]	
[ENOTAVAIL (page 547)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [EBADF] when *fildes* is a scan descriptor that was passed to one of the scan-related exit programs. See "Integrated File System Scan on Open Exit Program" on page 523 and "Integrated File System Scan on Close Exit Program" on page 513 for more information.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "close()—Close File or Socket Descriptor" on page 34—Close File or Socket Descriptor
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "dup2()—Duplicate Open File Descriptor to Another Descriptor" on page 58—Duplicate Open File Descriptor to Another Descriptor
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "open()—Open File" on page 195—Open File

Example

See Code disclaimer information for information pertaining to code examples.

The following example duplicates an open descriptor:

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <errno.h>

void print_file_id(int file_descriptor) {
  struct stat info;
  if (fstat(file_descriptor, &info) != 0)
    fprintf(stderr, "stat() error for file_descriptor %d: %s\n",
            strerror(errno));
  else
    printf("The file id of file_descriptor %d is %d\n",
           file_descriptor,(int) info.st_ino);
}

main() {
  int file_descriptor, file_descriptor2;
  char fn[]="original.file";

  /* create original file */
  if((file_descriptor = creat(fn,S_IRUSR | S_IWUSR)) < 0)
    perror("creat() error");
  /* generate a duplicate file descriptor of file_descriptor */
  else {
    if ((file_descriptor2 = dup(file_descriptor)) < 0)
      perror("dup() error");
    /* print resulting information */
    else {
      print_file_id(file_descriptor);
      print_file_id(file_descriptor2);
      puts("The file descriptors are different but");
      puts("they point to the same file.");
      close(file_descriptor);
      close(file_descriptor2);
```

```
    }
    unlink(fn);
  }
}
```

Output:

```
The file id of file_descriptor 0 is 30
The file id of file_descriptor 3 is 30
The file descriptors are different but
they point to the same file.
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

dup2()—Duplicate Open File Descriptor to Another Descriptor

Syntax

```
#include <unistd.h>

int dup2(int fildes, int fildes2);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 59.

The **dup2()** function returns a descriptor with the value *fildes2*. The descriptor refers to the same file as *fildes*, and it will close the file that *fildes2* was associated with. » For more information on the processing which may occur when the file is closed, see “close()—Close File or Socket Descriptor” on page 34—Close File or Socket Descriptor. «

If the original file descriptor was opened in text mode, data conversion is also done on the duplicated file descriptor.

The FD_CLOEXEC flag that is associated with the new file descriptor is cleared. Refer to “fcntl()—Perform File Control Command” on page 82—Perform File Control Command for additional information about the FD_CLOEXEC flag.

The following conditions apply:

- If *fildes2* is less than zero or greater than or equal to OPEN_MAX, **dup2()** returns -1 and sets the *errno* global variable to [EBADF].
- If *fildes* is a valid descriptor and is equal to *fildes2*, **dup2()** returns *fildes2* without closing it.
- If *fildes* is not a valid descriptor, **dup2()** fails and does not close *fildes2*.

This function works with descriptors for any type of object.

Parameters

fildes (Input) A descriptor to be duplicated.

fildes2 (Input) The descriptor to which the duplication is made.

Authorities

No authorization is required.

Return Value

value	**dup2()** was successful. The value of *fildes2* is returned.
-1	**dup2()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **dup2()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EIO (page 540)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. If this function is called by a thread executing one of the scan-related exit programs (or any of its created threads), it will fail with error code [ENOTSUP]. See "Integrated File System Scan on Open Exit Program" on page 523 and "Integrated File System Scan on Close Exit Program" on page 513 for more information.
2. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT

– Network File System
– QFileSvr.400

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "close()—Close File or Socket Descriptor" on page 34—Close File or Socket Descriptor
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "dup()—Duplicate Open File Descriptor" on page 55—Duplicate Open File Descriptor
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "open()—Open File" on page 195—Open File

Example

See Code disclaimer information for information pertaining to code examples.

The following example duplicates an open descriptor:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

void print_file_id(int file_descriptor) {
  struct stat info;
  if (fstat(file_descriptor, &info) != 0)
    fprintf(stderr, "stat() error for file_descriptor %d: %s\n",
            strerror(errno));
  else
    printf("The file id of file_descriptor %d is %d\n", file_descriptor,
           (int) info.st_ino);
}

main() {
  int file_descriptor, file_descriptor2;
  char fn[] = "original.file";
  char fn2[] = "dup2.file";

  /* create original file */
  if((file_descriptor = creat(fn, S_IRUSR | S_IWUSR)) < 0)
    perror("creat() error");
  /* create file to dup to */
  else if((file_descriptor2 = creat(fn2, S_IWUSR)) < 0)
    perror("creat()error");
  /* dup file_descriptor to file_descriptor2; print results */
  else {
    print_file_id(file_descriptor);
    print_file_id(file_descriptor2);
    if ((file_descriptor2 = dup2(file_descriptor, file_descriptor2)) < 0)
      perror("dup2() error");
    else {
      puts("After dup2()...");
      print_file_id(file_descriptor);
      print_file_id(file_descriptor2);
      puts("The file descriptors are different but they");
      puts("point to the same file which is different than");
      puts("the file that the second file_descriptor originally pointed to.");
      close(file_descriptor);
      close(file_descriptor2);
    }
```

```
    unlink(fn);
    unlink(fn2);
  }
}
```

Output:

```
The file id of file_descriptor 0 is 30
The file id of file_descriptor 3 is 58
After dup2()...
The file id of file_descriptor 0 is 30
The file id of file_descriptor 3 is 30
The file descriptors are different, but they
point to the same file, which is different than
the file that the second file_descriptor originally pointed to.
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

faccessx()—Determine File Accessibility for a Class of Users

Syntax

```
#include <unistd.h>

int faccessx(int fildes, int amode, int who);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 64.

The **faccessx()** function determines whether a file can be accessed by a specified class of users in a particular manner.

Adopted authority is not used.

Parameters

fildes (Input) The file descriptor of the file that is having its accessibility checked.

amode (Input) A bitwise representation of the access permissions to be checked.

The following symbols, which are defined in the <**unistd.h**> header file, can be used in *amode*:

F_OK (*x'00'*) Tests whether the file exists

R_OK (x'04') Tests whether the file can be accessed for reading

W_OK (*x'02'*) Tests whether the file can be accessed for writing

X_OK (*x'01'*) Tests whether the file can be accessed for execution

You can take the bitwise inclusive OR of any or all of the last three symbols to test several access modes at once. If you are using F_OK to test for the existence of the file, you cannot use OR with any of the other symbols. If any other bits are set in *amode*, **faccessx()** returns the [EINVAL] error.

who (Input) The class of users whose authority is to be checked.

The following symbols, which are defined in the <**unistd.h**> header file, can be used in *who*:

ACC_SELF
(*x'00'*) Determines if specified access is permitted for the current thread. The effective user and group IDs are used.

Note: If the real and effective user ID are the same and the real and effective group ID are the same, the request is treated as ACC_INVOKER. See the Usage Notes for more details.

ACC_INVOKER
(*x'01'*) Determines if specified access is permitted for the current thread. The effective user and group IDs are used.

ACC_OTHERS
(*x'08'*) Determines if specified access is permitted for any user other than the object owner. Only one of *R_OK*, *W_OK*, and *X_OK* is permitted when **who** is *ACC_OTHERS*. Privileged users (users with *ALLOBJ special authority) are not considered in this check.

ACC_ALL
(*x'20'*) Determines if specified access is permitted for all users. Only one of *R_OK*, *W_OK*, and *X_OK* is permitted when **who** is *ACC_ALL*. Privileged users (users with *ALLOBJ special authority) are not considered in this check.

Authorities

The following authorities are required if the **who** parameter is *ACC_SELF* or *ACC_INVOKER*. If *ACC_SELF* is specified, the effective UID and GID of the caller are used. If *ACC_INVOKER* is used, the real UID and GID of the caller are used.

Authorization Required for faccessx()

Object Referred to	Authority Required	errno
Object when R_OK is specified	*R	EACCES
Object when W_OK is specified	*W	EACCES
Object when X_OK is specified	*X	EACCES
Object when R_OK	W_OK is specified	*RW	EACCES
Object when R_OK	X_OK is specified	*RX	EACCES
Object when W_OK	X_OK is specified	*WX	EACCES
Object when R_OK	W_OK	X_OK is specified	*RWX	EACCES
Object when F_OK is specified	None	None

If the current thread has *ALLOBJ special authority, **faccessx()** with *ACC_SELF* or *ACC_INVOKER* will indicate success for R_OK, W_OK, or X_OK even if none of the permission bits are set.

Return Value

0 **faccessx()** was successful.

-1 **faccessx()** was not successful (or the specified access is not permitted for the class of users being checked). The *errno* global variable is set to indicate the error.

Error Conditions

If **faccessx()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	The class of users specified by the *who* parameter does not have the permission indicated by the *amode* parameter. If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EINTR (page 541)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETXTBSY (page 547)]	
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
   - Where multiple threads exist in the job.
   - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. ACC_SELF Mapped to ACC_INVOKER

   Some physical file systems do not support *ACC_SELF* for the *who* parameter. However, **faccessx()** will change the *who* parameter from *ACC_SELF* to *ACC_INVOKER* if the caller's real and effective user ID are equal, and the caller's real and effective group ID are equal.
3. Network File System Differences

   The Network File System will only support the value *ACC_INVOKER* for the *who* parameter. If **faccessx()** is called on a file in a mounted Network File System directory with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP. **Note:** If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)
4. QNTC File System Differences

   The QNTC File System will only support the value *ACC_INVOKER* for the *who* parameter. If **faccessx()** is called on a file in the QNTC File System with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP.

Note: If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

5. QOPT File System Differences

   If the file descriptor refers to an object that exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the object follows the rules described in the previous table, Authorization Required for faccessx() (page 62). If the object exists on a volume formatted in some other media format, no authorization checks are made on the object. The volume authorization list is checked for the requested authority regardless of the volume media format.

6. QFileSvr.400 File System Differences

   The QFileSvr.400 File System will only support the value *ACC_INVOKER* for the *who* parameter. If **faccessx()** is called on a file in the QFileSvr.400 File System with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP.

   Note: If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

7. QNetWare File System Differences

   The QNetWare File System will only support the value *ACC_INVOKER* for the *who* parameter. If **faccessx()** is called on a file in the QNetWare File System with a value for *who* other than *ACC_INVOKER*, the call will return -1 and errno ENOTSUP.

   Note: If the value for *who* has been mapped from *ACC_SELF* to *ACC_INVOKER* as previously described, then ENOTSUP will not be returned.

Related Information

- The <**unistd.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**limits.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “chmod()—Change File Authorizations” on page 22—Change File Authorizations
- “open()—Open File” on page 195—Open File
- “access()—Determine File Accessibility” on page 10—Determine File Accessibility
- “accessx()—Determine File Accessibility for a Class of Users” on page 14—Determine File Accessibility for a Class of Users
- “QlgAccessx()—Determine File Accessibility for a Class of Users (using NLS-enabled path name)” on page 237—Determine File Accessibility for a Class of Users (using NLS-enabled path name)
- “QlgAccess()—Determine File Accessibility (using NLS-enabled path name)” on page 235—Determine File Accessibility (using NLS-enabled path name)
- “stat()—Get File Information” on page 468—Get File Information

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines how a file is accessed:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

main() {
  char path[]="/myfile";
  int fd;

  fd = open(path, O_RDONLY);
  if (fd == -1)
  {
```

```
    printf("Error opening file.\n");
    return;
  }

  if (faccessx(fd, R_OK, ACC_OTHERS) == 0)
      printf("Someone besides the owner has read access to '%s'\n", path);
  if (faccessx(fd, W_OK, ACC_OTHERS) == 0)
      printf("Someone besides the owner has write access to '%s'\n", path);
  if (faccessx(fd, X_OK, ACC_OTHERS) == 0)
      printf("Someone besides the owner has search access to '%s'\n", path);
  close(fd);
}
```

Output:

In this example **faccessx()** was called on a descriptor for '/myfile'. The following would be the output if someone other than the owner has *R authority, someone besides the owner has *W authority, and noone other than the owner has *X authority.

```
Someone besides the owner has read access to '/'
Someone besides the owner has write access to '/'
```

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

---

fchdir()—Change Current Directory by Descriptor

Syntax

```
#include <unistd.h>

int fchdir(int fildes);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 68.

The **fchdir()** function makes the directory named by *fildes* the new current directory. If the **fchdir()** function fails, the current directory is unchanged.

Parameters

fildes (Input) The file descriptor of the directory.

Authorities

Note: Adopted authority is not used.

Authorization Required for fchdir()

Object Referred to	Authority Required	errno
» The directory named by *fildes*. «	*X	EACCES

Return Value

0 **fchdir()** was successful.

-1 **fchdir()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fchdir()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ENOENT (page 540)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400

   The **fchdir()** API operates on two objects: the previous current working directory and the new one. If either of these objects is managed by a file system that is not threadsafe, **fchdir()** fails with the ENOTSAFE error code.

2. Network File System Differences

   If the local storage of attributes and names is not suppressed (option noac when the file system is mounted), then one can potentially use the **fchdir()** API to change to a directory which has been removed. This depends on how often and when the local storage of attributes and names is refreshed.

Related Information

- The <**unistd.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**limits.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “chdir()—Change Current Directory” on page 19—Change Current Directory
- “getcwd()—Get Current Directory” on page 113—Get Current Directory
- “QlgChdir()—Change Current Directory (using NLS-enabled path name)” on page 239—Change Current Directory
- “QlgGetcwd()—Get Current Directory (using NLS-enabled path name)” on page 248—Get Current Directory

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **fchdir()**:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
```

```
main() {
  char dir[]="tempfile";
  int  file_descriptor;
  int oflag1 = O_RDONLY | O_CCSID;
  mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR;
  unsigned int open_ccsid = 37;

  if ((file_descriptor = open(dir,oflag1,mode,open_ccsid)) < 0)
    perror("open() error");
  else {
    if (fchdir(file_descriptor) != 0)
      perror("fchdir() to tempfile failed");
    close(file_descriptor);
  }
}
```

Output:

```
fchdir() to tempfile failed: Not a directory.
```

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

---

fchmod()—Change File Authorizations by Descriptor

Syntax

```
#include <sys/stat.h>

int fchmod(int fildes, mode_t mode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 26 for chmod().

The **fchmod()** function changes S_ISUID, S_ISGID, S_ISVTX, and the permission bits of the open file or directory, identified by *fildes*, to the corresponding bits specified in *mode*. **fchmod()** has no effect on file descriptions for files that are open at the time **fchmod()** is called.

fchmod() marks for update the change time of the file.

If the file is checked out by another user (someone other than the user profile of the current job), **fchmod()** fails with the [EBUSY] error.

Parameters

fildes (Input) The file descriptor of the file.

mode (Input) Bits that define S_ISUID, S_ISGID, S_ISVTX, and the access permissions of the file.

The *mode* argument is created with one of the symbols defined in the <**sys/stat.h**> header file. For more information on the symbols, refer to "chmod()—Change File Authorizations" on page 22.

If bits other than the bits listed above are set in *mode*, **fchmod()** returns the [EINVAL] error.

Authorities

Note: Adopted authority is not used.

Authorization Required for fchmod() (excluding QDLS)

Object Referred to	Authority Required	errno
Object	Owner (see **Note**)	EPERM
Note: You do not need the listed authority if you have *ALLOBJ special authority.		

Authorization Required for fchmod() in the QDLS File System

Object Referred to	Authority Required	errno
Object	Owner or *ALL	EACCES

Return Value

0 **fchmod()** was successful.

-1 **fchmod()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fchmod()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOSPC (page 541)]	
[ENOSYS (page 544)]	
[ENOSYSRSC (page 545)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	

Error condition	Additional information
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this API:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Related Information

- The <**sys/stat.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "chown()—Change Owner and Group of File" on page 29—Change Owner and Group of File
- "fchown()—Change Owner and Group of File by Descriptor" on page 72—Change Owner and Group of File by Descriptor
- "mkdir()—Make Directory" on page 169—Make Directory
- "open()—Open File" on page 195—Open File
- "stat()—Get File Information" on page 468—Get File Information

Example

See Code disclaimer information for information pertaining to code examples.

The following example changes a file permission:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

main() {
  char fn[]="temp.file";
  int  file_descriptor;
  struct stat info;

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    if (stat(fn, &info)!= 0)
       perror("stat() error");
    else {
       printf("original permissions were: %08o\n", info.st_mode);
    }
    if (fchmod(file_descriptor, S_IRWXU|S_IRWXG) != 0)
      perror("fchmod() error");
    else {
      if (stat(fn, &info)!= 0)
         perror("stat() error");
      else {
         printf("after fchmod(), permissions are: %08o\n", info.st_mode);
      }
    }
    if (close(file_descriptor)!= 0)
       perror("close() error");
    if (unlink(fn)!= 0)
       perror("unlink() error");
  }
}
```

Output:

```
original permissions were: 00100200
after fchmod(), permissions are: 00100770
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

fchown()—Change Owner and Group of File by Descriptor

Syntax

```
#include <unistd.h>

int fchown(int fildes, uid_t owner, gid_t group);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 75.

The **fchown()** function changes the owner and group of a file. The permissions of the previous owner or primary group to the object are revoked.

If the file is checked out by another user (someone other than the user profile of the current job), **fchown()** fails with the [EBUSY] error.

When **fchown()** completes successfully, it marks the change time of the file to be updated.

Parameters

fildes (Input) The file descriptor of the file.

owner (Input) The new user ID to be set for file.

group (Input) The new group ID to be set for file.

Note: Changing the owner or the primary group causes the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode to be cleared, unless the caller has *ALLOBJ special authority. If the caller does have *ALLOBJ special authority, the bits are not changed. This does not apply to directories, FIFO special files, or pipes. See the "chmod()—Change File Authorizations" on page 22 documentation.

Authorities

Note: Adopted authority is not used.

Authorization Required for fchown() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)

Object Referred to	Authority Required	errno
Object, when changing the owner	Owner and *OBJEXIST (also see **Note 1**)	EPERM
Object, when changing the primary group	See **Note 2**	EPERM
Previous owner's user profile, when changing the owner	*DLT	EPERM
New owner's user profile, when changing the owner	*ADD	EPERM
User profile of previous primary group, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM
Note: 1. You do not need the listed authority if you have *ALLOBJ special authority. 2. At least one of the following must be true: a. You have *ALLOBJ special authority. b. You are the owner and either of the following: • The new primary group is the primary group of the job. • The new primary group is one of the supplementary groups of the job.		

Authorization Required for fchown() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Object, when changing the owner	See **Note** (1)	EPERM
Object, when changing the primary group	See **Note** (2)	EPERM

Object Referred to	Authority Required	errno
Note: The required authorization varies for each object type. See the following commands in the iSeries Security Reference book for details: 1. CHGOBJOWN 2. CHGOBJPGP		

Authorization Required for fchown() in the QDLS File System

Object Referred to	Authority Required	errno
Object	*ALLOBJ Special Authority or Owner	EPERM
Previous owner's user profile, when changing the owner	*DLT	EPERM
New owner's user profile, when changing the owner	*ADD	EPERM
Previous primary group's user profile, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM

Authorization Required for fchown() in the QOPT File System

Object Referred to	Authority Required	errno
Volume authorization list	*CHANGE	EACCES
Each directory in the path name preceding the object.	*X	EACCES
Object	*ALLOBJ Special Authority or Owner	EPERM

Return Value

0 **fchown()** was successful.

-1 **fchown()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fchown()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	

Error condition	Additional information
[EINTR (page 541)]	
[EINVAL (page 540)]	For example, *owner* or *group* is not a valid user ID (UID) or group ID (GID). Or, the *owner* is the current primary group of the object.
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOSPC (page 541)]	
[ENOSYS (page 544)]	
[ENOSYSRSC (page 545)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     – ″Root″ (/)
     – QOpenSys
     – User-defined
     – QNTC
     – QSYS.LIB

- Independent ASP QSYS.LIB
- QOPT
- Network File System
- QFileSvr.400

2. QDLS File System Differences

   The owner and primary group of the /QDLS directory (root folder) cannot be changed. If an attempt is made to change the owner and primary group, a [ENOTSUP] error is returned.

3. QOPT File System Differences

   Changing the owner and primary group is allowed only for an object that exists on a volume formatted in Universal Disk Format (UDF). For all other media formats, ENOTSUP will be returned.

   QOPT file system objects that have owners will not be recognized by the Work with Objects by Owner (WRKOBJOWN) CL command. Likewise, QOPT objects that have a primary group will not be recognized by the Work Objects by Primary Group (WRKOBJPGP) CL command.

4. QFileSvr.400 File System Differences

   The QFileSvr.400 file system does not support **fchown()**.

5. QNetWare File System Differences

   Primary group is not supported. The GID must be zero on this API.

6. QNTC File System Differences

   The owner of files and directories cannot be changed. All files and directories in QNTC are owned by the QDFTOWN user profile.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chown()—Change Owner and Group of File" on page 29—Change Owner and Group of File
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "fchmod()—Change File Authorizations by Descriptor" on page 69—Change File Authorizations by Descriptor
- "mkdir()—Make Directory" on page 169—Make Directory
- "open()—Open File" on page 195—Open File
- "stat()—Get File Information" on page 468—Get File Information

Example

See Code disclaimer information for information pertaining to code examples.

The following example changes the owner ID and group ID:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

main() {
  char fn[]="temp.file";
  int  file_descriptor;
  struct stat info;

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    stat(fn, &info);
    printf("original owner was %d and group was %d\n", info.st_uid,
           info.st_gid);
    if (fchown(file_descriptor, 152, 0) != 0)
```

```
      perror("fchown() error");
    else {
      stat(fn, &info);
      printf("after fchown(), owner is %d and group is %d\n",
             info.st_uid, info.st_gid);
    }
    close(file_descriptor);
    unlink(fn);
  }
}
```

Output:

```
original owner was 137 and group was 0
after fchown(), owner is 152 and group is 0
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

fclear()—Write (Binary Zeros) to Descriptor

Syntax

```
#include <unistd.h>

off_t fclear
  (int file_descriptor, off_t nbyte);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 80.

The **fclear()** function writes *nbyte* bytes of binary zeros to the file associated with the *file_descriptor*. *nbyte* should not be greater than INT_MAX (defined in the <**limits.h**> header file). If it is, [EINVAL] will be returned. If *nbyte* is zero, **fclear()** simply returns a value of zero without attempting any other action.

If *file_descriptor* refers to a "regular file" (a stream file that can support positioning the file offset), **fclear()** begins writing binary zeros at the file offset associated with *file_descriptor*. A successful **fclear()** increments the file offset by the number of bytes written. If the incremented file offset is greater than the previous length of the file, the length of the file is set to the new file offset. An unsuccessful **fclear()** will not change the file offset. If the *file_descriptor* does not refer to a "regular file", [EINVAL] will be returned.

If O_APPEND (defined in the <**fcntl.h**> header file) is set for the file, **fclear()** does **not** set the file offset to the end of the file before writing the output. Instead, it begins writing binary zeros at the current file offset associated with the *file_descriptor*.

If **fclear()** is called such that *nbyte* plus the current file offset will cause the size of the file to exceed 2GB minus 1 bytes when the file is **not** opened for large file access, the system allowed maximum file size when the file is opened for large file access, or the process soft file size limit, [EFBIG] will be returned.

If **fclear()** is successful and *nbyte* is greater than zero, the change and modification times for the file are updated.

If *file_descriptor* refers to a descriptor obtained using the **open()** function with O_TEXTDATA specified, binary zeros are written to the file assuming they are in textual form. The data (binary zeros) is converted from the code page of the application, job, or system, to the code page of the file as follows:

- Only simple conversions are performed. That is, if one byte of binary zeros does not convert to one byte of binary zeros then [ENOTSUP] is returned.

- When clearing a physical file in the QSYS.LIB file system the **fclear()** should not be performed across a record boundary. If it is, [ETRUNC] will be returned.

Note: The conversion of binary zeros will always result in binary zeros.

There are some important considerations if O_CCSID was specified on the **open()**.

- If an **fclear()** is performed when there are partial characters buffered internally due to a previous **write()**, the **fclear()** will fail with the [ENOTSUP] error.
- Because of the above consideration and because of the possible expansion or contraction of converted data, applications using the O_CCSID flag should avoid assumptions about data size and the current file offset.

If O_TEXTDATA was not specified on the **open()**, binary zeros are written to the file without conversion. The application is responsible for handling the data.

Note: When the **fclear** completes successfully, the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode will be cleared. If the **fclear()** is unsuccessful, the bits are undefined.

Parameters

file_descriptor
(Input) The descriptor of the file to be cleared (write binary zeros).

nbyte
(Input) Indicates the number of bytes to clear (write binary zeros).

Authorities

No authorization is required.

Return Value

value	**fclear()** was successful. The return value is the number of bytes that have been successfully cleared. This number will be equal to *nbyte*.
-1	**fclear()** was not successful. The **fclear()** was not able to clear all of the bytes requested. No indication is given as to how much data was successfully cleared. In addition, the file offset will remaind unchanged. The *errno* global variable is set to indicate the error.

Error Conditions

If **fclear()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	**Additional information**
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	

Error condition	Additional information
[EFBIG (page 545)]	The size of the file would exceed 2GB minus 1 bytes when the file is **not** opened for large file access, the system allowed maximum file size when the file is opened for large file access, or the process soft file size limit.
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ENXIO (page 541)]	
[ERESTART (page 547)]	
[ETRUNC (page 540)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

Additionally, if interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - QFileSvr.400
     - Network File System
2. QSYS.LIB and independent ASP QSYS.LIB File System Differences

   This function will fail with error code [ENOTSAFE] if the object on which this function is operating is a save file and multiple threads exist in the job.

   An **fclear()** request should not be done on a save file. If it is, unpredictable results may occur.

   A successful **fclear()** updates the change, modification, and access times for a database member using the normal rules that apply to database files. At most, the access time is updated once per day.
3. QOPT File System Differences

   The change and modification times of the file are updated when the file is closed.

   When writing to files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being cleared are ignored.
4. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations (several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data).

   Reading, writing, and clearing of files with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the "fcntl()—Perform File Control Command" on page 82 API to get and release these locks.
5. QFileSvr.400 File System Differences

   This file system does not support **fclear()** or **fclear64()**, [ENOTSUP] will be returned.
6. QNTC File System Differences

   This file system does not support **fclear()** or **fclear64()**, [ENOTSUP] will be returned.
7. File System Differences

   File systems other than "root" (/), QOpenSys, user-defined, QNTC and QFileSvr.400 will be restricted to doing **fclear()**s no larger than 2GB minus 1 bytes. If this rule is violated [EINVAL] will be returned.

8. For the file systems that do not support large files, **fclear()** will return [EINVAL] if *nbyte* plus the file offset exceeds 2GB minus 1 bytes, regardless of how the file was opened. For the file systems that do support large files, **fclear()** will return [EFBIG] if *nbyte* plus the file offset exceeds 2GB minus 1 bytes and the file was not opened for large file access.
9. If the **fclear()** exceeds the process soft file size limit, signal SIFXFSZ is issued.

Related Information

- The <**fcntl.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "dup()—Duplicate Open File Descriptor" on page 55—Duplicate Open File Descriptor
- "dup2()—Duplicate Open File Descriptor to Another Descriptor" on page 58—Duplicate Open File Descriptor to Another Descriptor
- "fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)" on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "ftruncate()—Truncate File" on page 109—Truncate File
- "ftruncate64()—Truncate File (Large File Enabled)" on page 113—Truncate File (Large File Enabled)
- "ioctl()—Perform I/O Control Request" on page 141—Perform I/O Control Request
- "lseek()—Set File Read/Write Offset" on page 157—Set File Read/Write Offset
- "open()—Open File" on page 195—Open File
- "pread()—Read from Descriptor with Offset" on page 223—Read from Descriptor with Offset
- "pread64()—Read from Descriptor with Offset (large file enabled)" on page 228—Read from Descriptor with Offset (large file enabled)
- "pwrite()—Write to Descriptor with Offset" on page 229—Write to Descriptor with Offset
- "pwrite64()—Write to Descriptor with Offset (large file enabled)" on page 234—Write to Descriptor with Offset (large file enabled)
- "read()—Read from Descriptor" on page 437—Read from Descriptor
- "readv()—Read from Descriptor Using Multiple Buffers" on page 455—Read from Descriptor Using Multiple Buffers
- "write()—Write to Descriptor" on page 502—Write to Descriptor
- "writev()—Write to Descriptor Using Multiple Buffers" on page 509—Write to Descriptor Using Multiple Buffers

Example

See Code disclaimer information for information pertaining to code examples.

The following example clears a specific number of bytes in a file:

```
#include <unistd.h>
#include <stdio.h>

main() {
  int  fileDescriptor;
  off_t  ret;
  int  oflags = O_CREAT | O_RDWR;
  mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR;

  if ((fileDescriptor = open("foo", oflags, mode)) < 0)
    perror("open() error");
  else {
    if ((ret = fclear(fileDescriptor,  10)) == -1)
      perror("fclear() error");
```

```
    else printf("fclear() cleared %d bytes.\n", ret);
    if (close(fileDescriptor)!= 0)
       perror("close() error");
    if (unlink("foo")!= 0)
       perror("unlink() error");
  }
}
```

Output:

```
fclear() cleared 10 bytes.
```

API introduced: V5R3

Top | UNIX-Type APIs | APIs by category

fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)

Syntax

```
#include <unistd.h>

off64_t fclear
  (int file_descriptor, off64_t nbyte);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes.

The **fclear64()** function writes *nbyte* bytes of binary zeros to the file associated with the *file_descriptor*. If *nbyte* is zero, **fclear64()** simply returns a value of zero without attempting any other action.

fclear64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 bytes and is capable of clearing up to the system allowed maximum file size bytes as long as the file exists in Root, QOpenSys, and UDFS file systems and has been opened by either of the following:

- Using the **open64()** function (see "open64()—Open File (Large File Enabled)" on page 211).
- Using the **open()** function (see "open()—Open File" on page 195) with the O_LARGEFILE flag set in the oflag parameter.

For additional information about parameters, authorities, error conditions, and examples, see "fclear()—Write (Binary Zeros) to Descriptor" on page 77.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **fclear64()** API, you must compile the source with the _LARGE_FILE_API macro defined.
2. All of the usage notes for **fclear()** apply to **fclear64()**. See *Usage Notes* in the **fclear()** API.

API introduced: V5R3

Top | UNIX-Type APIs | APIs by category

fcntl()—Perform File Control Command

Syntax

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl(int descriptor,
          int command,
          ...)
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 90.

The **fcntl()** function performs various actions on open descriptors, such as obtaining or changing the attributes of a file or socket descriptor.

Parameters

descriptor
(Input) The descriptor on which the control command is to be performed, such as having its attributes retrieved or changed.

command
(Input) The command that is to be performed on the *descriptor*.

...
(Input) A variable number of optional parameters that is dependent on the *command*. Only some of the commands use this parameter.

The *fcntl()* commands that are supported are:

F_DUPFD
Duplicates the descriptor. A third **int** argument must be specified. **fcntl()** returns the lowest descriptor greater than or equal to this third argument that is not already associated with an open file. This descriptor refers to the same object as *descriptor* and shares any locks. If the original descriptor was opened in text mode, data conversion is also done on the duplicated descriptor. The FD_CLOEXEC flag that is associated with the new descriptor is cleared.

F_GETFD
Obtains the descriptor flags for *descriptor*. **fcntl()** returns these flags as its result. For a list of supported file descriptor flags, see "Flags" on page 84. Descriptor flags are associated with a single descriptor and do not affect other descriptors that refer to the same object.

F_GETFL
Obtains the » open « flags for *descriptor*. **fcntl()** returns these flags as its result. For a list of » the open « flags, see "Using the oflag Parameter" on page 197 in **open()**.

F_GETLK
Obtains locking information for an object. You must specify a third argument of type `struct flock *`. See "File Locking" on page 85 for details. **fcntl()** returns 0 if it successfully obtains the locking information. When you develop in C-based languages and the function is compiled with the _LARGE_FILES macro defined, F_GETLK is mapped to the F_GETLK64 symbol.

F_GETLK64
Obtains locking information for a large file. You must specify a third argument of type `struct flock64 *`. See "File Locking" on page 85 for details. **fcntl()** returns 0 if it successfully obtains the locking information. When you develop in C-based languages, it is necessary to compile the function with the _LARGE_FILE_API macro defined to use this symbol.

F_GETOWN
Returns the process ID or process group ID that is set to receive the `SIGIO` (I/O is possible on a descriptor) and `SIGURG` (urgent condition is present) signals. For more information, see Signal APIs.

F_SETFD
Sets the descriptor flags for *descriptor*. You must specify a third *int* argument, which gives the new file descriptor flag settings (see "Flags" on page 84). If any other bits in the third argument are set, **fcntl()** fails with the [EINVAL] error. **fcntl()** returns 0 if it successfully sets the flags. Descriptor flags are associated with a single descriptor and do not affect other descriptors that refer to the same object.

F_SETFL
Sets status flags for the descriptor. You must specify a third *int* argument, giving the new file status flag settings (see "Flags" on page 84). **fcntl()** does not change the file access mode, and file access bits in the third argument are ignored. All other oflag values that are valid on the **open()** API are also ignored. If any other bits in the third argument are set, **fcntl()** fails with the [EINVAL] error. **fcntl()** returns `0` if it successfully sets the flags.

F_SETLK
Sets or clears a file segment lock. You must specify a third argument of type `struct flock *`. See "File Locking" on page 85 for details. **fcntl()** returns 0 if it successfully clears the lock. When you develop in C-based languages and the function is compiled with the _LARGE_FILES macro defined, F_SETLK is mapped to the F_SETLK64 symbol.

F_SETLK64 — Sets or clears a file segment lock for a large file. You must specify a third argument of type `struct flock64 *`. See "File Locking" on page 85 for details. **fcntl()** returns 0 if it successfully clears the lock. When you develop in C-based languages, it is necessary to compile the function with the _LARGE_FILE_API macro defined to use this symbol.

F_SETLKW — Sets or clears a file segment lock; however, if a shared or exclusive lock is blocked by other locks, **fcntl()** waits until the request can be satisfied. You must specify a third argument of type `struct flock *`. See "File Locking" on page 85 for details. When you develop in C-based languages and the function is compiled with the _LARGE_FILES macro defined, F_SETLKW is mapped to the F_SETLKW64 symbol.

F_SETLKW64 — Sets or clears a file segment lock on a large file; however, if a shared or exclusive lock is blocked by other locks, **fcntl()** waits until the request can be satisfied. See "File Locking" on page 85 for details. You must specify a third argument of type `struct flock64 *`. When you develop in C-based languages, it is necessary to compile the function with the _LARGE_FILE_API macro defined to use this symbol.

F_SETOWN — Sets the process ID or process group ID that is to receive the `SIGIO` and `SIGURG` signals. For more information, see Signal APIs.

Flags

There are several types of flags associated with each open object. Flags for an object are represented by symbols defined in the <**fcntl.h** header file. The following *file status* flags can be associated with an object:

FASYNC — The `SIGIO` signal is sent to the process when it is possible to do I/O.
» This function will fail with error code [EINVAL] when *fildes* is for an object other than a socket.
«

FNDELAY — This flag is defined to be equivalent to `O_NDELAY`.

O_APPEND — Append mode. If this flag is 1, every write operation on the file begins at the end of the file.

O_DSYNC — Synchronous update - data only. If this flag is 1, all file data is written to permanent storage before the update operation returns. Update operations include, but are not limited to, the following: **ftruncate()**, **open()** with O_TRUNC, and **write()**.

O_NDELAY — This flag is defined to be equivalent to `O_NONBLOCK`.

O_NONBLOCK — Non-blocking mode. If this flag is 1, read or write operations on the file will not cause the thread to block. This file status flag applies only to pipe, FIFO, and socket descriptors.

O_RSYNC — Synchronous read. If this flag is 1, read operations to the file will be performed synchronously. This flag is used in combination with O_SYNC or O_DSYNC. When O_RSYNC and O_SYNC are set, all file data and file attributes are written to permanent storage before the read operation returns. When O_RSYNC and O_DSYNC are set, all file data is written to permanent storage before the read operation returns.

O_SYNC — Synchronous update. If this flag is 1, all file data and file attributes relative to the I/O operation are written to permanent storage before the update operation returns. Update operations include, but are not limited to, the following: **ftruncate()**, **open()** with O_TRUNC, and **write()**.

The following *file access mode* flags can be associated with a file:

O_RDONLY — The file is opened for reading only.

O_RDWR — The file is opened for reading and writing.

O_WRONLY — The file is opened for writing only.

A mask can be used to extract flags:

O_ACCMODE — Extracts file access mode flags.

The following *descriptor* flags can be associated with a descriptor:

FD_CLOEXEC
: Controls descriptor inheritance during **spawn()** and **spawnp()** when simple inheritance is being used, as follows:

- If the FD_CLOEXEC flag is zero, the descriptor is inherited by the child process that is created by the **spawn()** or **spawnp()**API.

  Note: Descriptors that are created as a result of the **opendir()** API (to implement open directory streams) are not inherited, regardless of the value of the FD_CLOEXEC flag.
- If the FD_CLOEXEC flag is set, the descriptor is not inherited by the child process that is created by the **spawn()** or **spawnp()** API.

Refer to spawn()—Spawn Process and spawnp()—Spawn Process with Path for additional information about FD_CLOEXEC.

File Locking

A local or remote job can use **fcntl()** to lock out other local or remote jobs from a part of a file. By locking out other jobs, the job can read or write to that part of the file without interference from others. File locking can ensure data integrity when several jobs have a file accessed concurrently. For more information about remote locking, see information about the network lock manager and the network status monitor in the Network File System Support book.

All locks obtained using **fcntl()** are advisory only. Jobs can use advisory locks to inform each other that they want to protect parts of a file, but advisory locks do not prevent input and output on the locked parts. If a job has appropriate permissions on a file, it can perform whatever I/O it chooses, regardless of what advisory locks are set. Therefore, advisory locking is only a convention, and it works only when all jobs respect the convention.

Another type of lock, called a mandatory lock, can be set by a remote personal computer application. Mandatory locks restrict I/O on the locked parts. A read fails when reading a part that is locked with a mandatory write lock. A write fails when writing a part that is locked with a mandatory read or mandatory write lock.

Two different structures are used to control locking operations: `struct flock` and `struct flock64` (both defined in the <**fcntl.h** header file). You can use `struct flock64` with the F_GETLK64, F_SETLK64, and F_SETLKW64 commands to control locks on large files (files greater than 2GB minus 1 byte). The `struct flock` structure has the following members:

short	l_type	Indicates the type of lock, as indicated by one of the following symbols (defined in the <**fcntl.h**> header file): *F_RDLCK* Indicates a *read lock*; also called a *shared lock*. When a job has a read lock, no other job can obtain write locks for that part of the file. More than one job can have a read lock on the same part of a file simultaneously. To establish a read lock, a job must have the file accessed for reading. *F_WRLCK* Indicates a *write lock*; also called an *exclusive lock*. When a job has a write lock, no other job can obtain a read lock or write lock on the same part or an overlapping part of that file. A job cannot put a write lock on part of a file if another job already has a read lock on an overlapping part of the file. To establish a write lock, a job must have accessed the file for writing. *F_UNLCK* Unlocks a lock that was set previously.

short	l_whence	One of three symbols used in determining the part of the file that is affected by this lock. These symbols are defined in the <**unistd.h**> header file and are the same as symbols used by **lseek()**: *SEEK_CUR* The current file offset in the file. *SEEK_END* The end of the file. *SEEK_SET* The start of the file.
off_t	l_start	Gives a byte offset used to identify the part of the file that is affected by this lock. If `l_start` is negative, it is handled as an unsigned value. The part of the file affected by the lock begins at this offset from the location given by `l_whence`. For example, if `l_whence` is SEEK_SET and `l_start` is 10, the locked part of the file begins at an offset of 10 bytes from the beginning of the file.
off_t	l_len	Gives the size of the locked part of the file, in bytes. If the size is negative, it is treated as an unsigned value. If `l_len` is zero, the locked part of the file begins at the position specified by `l_whence` and `l_start`, and extends to the end of the file. Together, `l_whence`, `l_start`, and `l_len` are used to describe the part of the file that is affected by this lock.
pid_t	l_pid	Specifies the job ID of the job that holds the lock. This is an output field used only with F_GETLK actions.
void	*l_reserved0	Reserved. Must be set to NULL.
void	*l_reserved1	Reserved. Must be set to NULL.

When you develop in C-based languages and this function is compiled with _LARGE_FILES defined, the `struct flock` data type will be mapped to a `struct flock64` data type. To use the `struct flock64` data type explicitly, it is necessary to compile the function with _LARGE_FILE_API defined.

The `struct flock64` structure has the following members:

short	l_type	Indicates the type of lock, as indicated by one of the following symbols (defined in the <**fcntl.h** header file): *F_RDLCK* Indicates a *read lock*; also called a *shared lock*. When a job has a read lock, no other job can obtain write locks for that part of the file. More than one job can have a read lock on the same part of a file simultaneously. To establish a read lock, a job must have the file accessed for reading. *F_WRLCK* Indicates a *write lock*; also called an *exclusive lock*. When a job has a write lock, no other job can obtain a read lock or write lock on the same part or an overlapping part of that file. A job cannot put a write lock on part of a file if another job already has a read lock on an overlapping part of the file. To establish a write lock, a job must have accessed the file for writing. *F_UNLCK* Unlocks a lock that was set previously.

short	l_whence	One of three symbols used in determining the part of the file that is affected by this lock. These symbols are defined in the <**unistd.h**> header file and are the same as symbols used by **lseek()**: *SEEK_CUR* The current file offset in the file. *SEEK_END* The end of the file. *SEEK_SET* The start of the file.
char	l_reserved2[4]	Reserved field
off64_t	l_start	Gives a byte offset used to identify the part of the file that is affected by this lock. `l_start` is handled as a signed value. The part of the file affected by the lock begins at this offset from the location given by `l_whence`. For example, if `l_whence` is SEEK_SET and `l_start` is 10, the locked part of the file begins at an offset of 10 bytes from the beginning of the file.
off64_t	l_len	Gives the size of the locked part of the file, in bytes. If the size is negative, the part of the file affected is l_start + l_len through l_start - 1. If `l_len` is zero, the locked part of the file begins at the position specified by `l_whence` and `l_start`, and extends to the end of the file. Together, `l_whence`, `l_start`, and `l_len` are used to describe the part of the file that is affected by this lock.
pid_t	l_pid	Specifies the job ID of the job that holds the lock. This is an output field used only with F_GETLK actions.
char	reserved3[4]	Reserved field.
void	*l_reserved0	Reserved. Must be set to NULL.
void	*l_reserved1	Reserved. Must be set to NULL.

You can set locks by specifying F_SETLK or F_SETLK64 as the *command* argument for **fcntl()**. Such a function call requires a third argument pointing to a `struct flock` structure (or `struct flock64` in the case of F_SETLK64), as in this example:

```
struct flock lock_it;
lock_it.l_type = F_RDLCK;
lock_it.l_whence = SEEK_SET;
lock_it.l_start = 0;
lock_it.l_len = 100;
fcntl(file_descriptor,F_SETLK,&lock_it);
```

This example sets up a flock structure describing a read lock on the first 100 bytes of a file, and then calls **fcntl()** to establish the lock. You can unlock this lock by setting `l_type` to F_UNLCK and making the same call. If an F_SETLK operation cannot set a lock, it returns immediately with an error saying that the lock cannot be set.

The F_SETLKW and F_SETLKW64 operations are similar to F_SETLK and F_SETLK64, except that they wait until the lock can be set. For example, if you want to establish an exclusive lock and some other job already has a lock established on an overlapping part of the file, **fcntl()** waits until the other process has removed its lock.

F_SETLKW and F_SETLKW64 operations can encounter *deadlocks* when job A is waiting for job B to unlock a region and job B is waiting for job A to unlock a different region. If the system detects that an F_SETLKW or F_SETLKW64 might cause a deadlock, **fcntl()** fails with *errno* set to [EDEADLK].

With the F_SETLK64, F_SETLKW64, and F_GETLK64 operations, the maximum offset that can be specified is the largest value that can be held in an 8-byte, signed integer.

A job can determine locking information about a file by using F_GETLK and F_GETLK64 as the *command* argument for **fcntl()**. In this case, the call to **fcntl()** should specify a third argument pointing to a flock structure. The structure should describe the lock operation you want. When **fcntl()** returns, the structure indicated by the flock pointer is changed to show the first lock that would prevent the proposed lock operation from taking place. The returned structure shows the type of lock that is set, the part of the file that is locked, and the job ID of the job that holds the lock. In the returned structure:

- `l_whence` is always SEEK_SET.
- `l_start` gives the offset of the locked portion from the beginning of the file.
- `l_len` is the length of the locked portion.

If there are no locks that prevent the proposed lock operation, the returned structure has F_UNLCK in `l_type` and is otherwise unchanged.

If **fcntl()** attempts to operate on a large file (one larger than 2GB minus 1 byte) with the F_SETLK, F_GETLK, or FSETLKW commands, the API fails with [EOVERFLOW]. To work with large files, compile with the _LARGE_FILE_API macro defined (when you develop in C-based languages) and use the F_SETLK64, F_GETLK64, or FSETLKW64 commands. When you develop in C-based languages, it is also possible to work with large files by compiling the source with the _LARGE_FILES macro label defined. Note that the file must have been opened for large file access (either the **open64()** API was used or the **open()** API was used with the O_LARGEFILE flag defined in the oflag parameter).

An application that uses the F_SETLK or F_SETLKW commands may try to lock or unlock a file that has been extended beyond 2GB minus 1 byte by another application. If the value of l_len is set to 0 on the lock or unlock request, the byte range held or released will go to the end of the file rather than ending at offset 2GB minus 2.

An application that uses the F_SETLK or F_SETLKW commands also may try to lock or unlock a file that has been extended beyond offset 2GB minus 2 with l_len NOT set to 0. If this application attempts to lock or unlock the byte range up to offset 2GB minus 2 and l_len is not 0, the unlock request will unlock the file only up to offset 2GB minus 2 rather than to the end of the file.

A job can have several locks on a file at the same time, but only one type of lock can be set on a given byte. Therefore, if a job puts a new lock on a part of a file that it had locked previously, the job has only one lock on that part of the file. The type of the lock is the one specified in the most recent locking operation.

Locks can start and extend beyond the current end of a file, but cannot start or extend ahead of the beginning of a file.

All of the locks a job has on a file are removed when the job closes any descriptor that refers to the locked file.

The maximum starting offset that can be specified by using the fnctl() API is 2^{63} - 1, the largest number that can be represented by a signed 8-byte integer. Mandatory locks set by a personal computer application or by a user of the DosSetFileLocks64() API may lock a byte range that is greater than 2^{63} - 1.

An application that uses the F_SETLK64 or F_SETLKW64 commands can lock the offset range that is beyond 2^{63} - 1 by locking offset 2^{63} - 1. When offset 2^{63} - 1 is locked, it implicitly locks to the end of the file. The end of the file is the largest number than can be represented by an 8-byte unsigned integer or 2^{64} - 1. This implicit lock may inhibit the personal computer application from setting mandatory locks in the range not explicitly accessable by the fcntl() API.

Any lock set using the fcntl() API that locks offset 2^{63} - 1 will have a length of 0.

An application that uses the F_GETLK64 may encounter a mandatory lock set by a personal computer application, which locks a range of offsets greater than 2^{63} - 1. This lock conflict will have a starting offset equal to or less than 2^{63} - 1 and a length of 0.

Authorities

No authorization is required.

Return Value

value	**fcntl()** was successful. The value returned depends on the *command* that was specified.
-1	**fcntl()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fcntl()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	The process tried to lock with F_SETLK, but the lock is in conflict with a previously established lock.
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBADFUNC (page 540)]	A given descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open object.
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EDEADLK (page 543)]	
[EFAULT (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EMFILE (page 543)]	
[ENOLCK (page 544)]	
[ENOMEM (page 543)]	
[ENOSYS (page 544)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[EOVERFLOW (page 546)]	One of the values to be returned cannot be represented correctly. The command argument is F_GETLK, F_SETLK, or F_SETLKW and the offset of any byte in the requested segment cannot be represented correctly in a variable of type off_t (the offset is greater than 2GB minus 1 byte).
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPFA0D4 E	File system error occurred. Error number &1.
CPFA081 E	Unable to set return value or error code.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. If F_DUPFD is specified as the **fcntl()** command, this function will fail with error code [EBADF] when *fildes* is a scan descriptor that was passed to one of the scan-related exit programs. See "Integrated File System Scan on Open Exit Program" on page 523 and "Integrated File System Scan on Close Exit Program" on page 513 for more information.

3. If the **fcntl()** command is called by a thread executing one of the scan-related exit programs (or any of its created threads), it will fail with error code [ENOTSUP] if F_SETLK, F_SETLK64, F_SETLKW or F_SETLKW64 is specified. See “Integrated File System Scan on Open Exit Program” on page 523 and “Integrated File System Scan on Close Exit Program” on page 513 for more information.
4. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   The following **fcntl()** commands are not supported:
   - F_GETLK
   - F_SETLK
   - F_SETLKW

   Using any of these commands results in an [ENOSYS] error.
5. Network File System Differences

   Reading and writing to a file with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the **fcntl()** API to get and release these locks. For more information about remote locking, see information about the network lock manager and the network status monitor in the Network File System Support book.
6. QNetWare File System Differences

   F_GETLK and F_SETLKW are not supported. F_RDLCK and F_WRLCK are ignored. All locks prevent reading and writing. Advisory locks are not supported. All locks are mandatory locks. Locking a file that is opened more than once in the same job with the same access mode is not supported, and its result is undefined.
7. This function will fail with the [EOVERFLOW] error if the command is F_GETLK, F_SETLK, or F_SETLKW and the offset or the length exceeds offset 2 GB minus 2.
8. When you develop in C-based languages and an application is compiled with the _LARGE_FILES macro defined, the `struct flock` data type will be mapped to a `struct flock64` data type. To use the `struct flock64` data type explicitly, it is necessary to compile the function with the _LARGE_FILE_API defined.
9. In several cases, similar function can be obtained by using *ioctl()*.

Related Information

- The <**sys/types.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**unistd.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**fcntl.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “close()—Close File or Socket Descriptor” on page 34—Close File or Socket Descriptor
- “dup()—Duplicate Open File Descriptor” on page 55—Duplicate Open File Descriptor
- “dup2()—Duplicate Open File Descriptor to Another Descriptor” on page 58—Duplicate Open File Descriptor to Another Descriptor
- “ioctl()—Perform I/O Control Request” on page 141—Perform I/O Control Request
- “lseek()—Set File Read/Write Offset” on page 157—Set File Read/Write Offset
- “open()—Open File” on page 195—Open File
- spawn()—Spawn Process
- spawnp()—Spawn Process with Path
- Network File System Support book

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **fcntl()**:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
  int flags;
  int append_flag;
  int nonblock_flag;
  int access_mode;
  int file_descriptor; /* File Descriptor */
  char *text1 = "abcdefghij";
  char *text2 = "0123456789";
  char read_buffer[25];

  memset(read_buffer, '\0', 25);

  /* create a new file */
  file_descriptor = creat("testfile",S_IRWXU);
  write(file_descriptor, text1, 10);
  close(file_descriptor);

  /* open the file with read/write access */
  file_descriptor = open("testfile", O_RDWR);
  read(file_descriptor, read_buffer,24);
  printf("first read is \'%s\'\n",read_buffer);

  /* reset file pointer to the beginning of the file */
  lseek(file_descriptor, 0, SEEK_SET);
  /* set append flag to prevent overwriting existing text */
  fcntl(file_descriptor, F_SETFL, O_APPEND);
  write(file_descriptor, text2, 10);
  lseek(file_descriptor, 0, SEEK_SET);
  read(file_descriptor, read_buffer,24);
  printf("second read is \'%s\'\n",read_buffer);

  close(file_descriptor);
  unlink("testfile");

  return 0;
}
```

Output:

```
first read is 'abcdefghij'
second read is 'abcdefghij0123456789'
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

fpathconf()—Get Configurable Path Name Variables by Descriptor

Syntax

```
#include <unistd.h>

long fpathconf(int file_descriptor, int name);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 94.

The **fpathconf()** function determines the value of a configuration variable (*name*) associated with a particular file descriptor (*file_descriptor*). **fpathconf()** works exactly like **pathconf()**, except that it takes a file descriptor as an argument rather than taking a path name.

If *file_descriptor* is a descriptor for a socket, **fpathconf()** returns an error of [EINVAL].

Parameters

file_descriptor
(Input) A file descriptor of the file for which the value of the configurable variable is requested.

name (Input) The name of the configuration variable value requested.

The value of *name* can be any one of a set of symbols defined in the <**unistd.h**> include file. For more information, see "pathconf()—Get Configurable Path Name Variables" on page 216

Authorities

No authorization is required.

Return Value

value **fpathconf()** was successful. The value of the variable requested in *name* is returned.

-1 One of the following has occurred:

- A particular variable has no limit (for example, _PC_PATH_MAX). The *errno* global variable is not changed.
- **fpathconf()** was not successful. The *errno* is set.

Error Conditions

If **fpathconf()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EFAULT (page 541)]	
[EINVAL (page 540)]	For example, *name* is not a valid configuration variable name, or the given variable cannot be associated with the specified file.
[EIO (page 540)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

Error condition	Additional information
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System

– QFileSvr.400

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "open()—Open File" on page 195—Open File
- "pathconf()—Get Configurable Path Name Variables" on page 216—Get Configurable Path Name Variables
- "QlgPathconf()—Get Configurable Path Name Variables (using NLS-enabled path name)" on page 278—Get Configurable Path Name Variables

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **fpathconf()**:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

main() {
  long result;
  char fn[]="temp.file";
  int  file_descriptor;

  if ((file_descriptor = creat(fn, S_IRUSR)) < 0)
    perror("creat() error");
  else {
    errno = 0;
    puts("examining NAME_MAX limit for current working directory's");
    puts("filesystem:");
    if ((result = fpathconf(file_descriptor, _PC_NAME_MAX)) == -1)
      if (errno == 0)
        puts("There is no limit to NAME_MAX.");
      else perror("fpathconf() error");
    else
      printf("NAME_MAX is %ld\n", result);
    close(file_descriptor);
    unlink(fn);
  }
}
```

Output:

```
examining NAME_MAX limit for current working directory's
filesystem:
NAME_MAX is 255
```

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

---

fstat()—Get File Information by Descriptor

Syntax

```
#include <sys/stat.h>

int fstat(int descriptor,
          struct stat *buffer)
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 97.

The **fstat()** function gets status information about the object specified by the open descriptor *descriptor* and stores the information in the area of memory indicated by the *buffer* argument. The status information is returned in a `stat` structure, as defined in the <**sys/stat.h**> header file.

Parameters

descriptor
(Input) The descriptor for which information is to be retrieved.

buffer
(Output) A pointer to a buffer of type **struct stat** in which the information is returned. The structure pointed to by the *buffer* parameter is described in "stat()—Get File Information" on page 468.

The *st_mode*, *st_dev*, and *st_blksize* fields are the only fields set for socket descriptors. The *st_mode* field is set to a value that indicates the descriptor is a socket descriptor, the *st_dev* field is set to -1, and the *st_blksize* field is set to an optimal value determined by the system.

Authorities

No authorization is required.

Return Value

0	**fstat()** was successful. The information is returned in *buffer*.
-1	**fstat()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fstat()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBADFUNC (page 540)]	A given descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open object.
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EINVAL (page 540)]	This error code may be returned when the underlying object represented by the descriptor is unable to fill the **stat** structure (for example, if the function was issued against a socket descriptor that had its connection reset).
[EIO (page 540)]	
[ENOBUFS (page 542)]	
[ENOSYSRSC (page 545)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	

Error condition	Additional information
[EOVERFLOW (page 546)]	The specified file exists and its size is too large to be represented in the structure pointed to by *buffer* (the file is larger than 2GB minus 1 byte).
[EPERM (page 540)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNATCH (page 543)]	
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ETIMEDOUT (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPFA0D4 E	File system error occurred. Error number &1.
CPFA081 E	Unable to set return value or error code.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
   - Where multiple threads exist in the job.
   - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT

– Network File System
– QFileSvr.400

2. Sockets-Specific Notes
   - The field *st_mode* can be inspected using the S_ISSOCK macro (defined in **<sys/stat.h>**) to determine if the descriptor is pointing to a socket descriptor.
   - For socket descriptors, use the send buffer size (this is the value returned for *st_blksize*) for the length parameter on your input and output functions. This can improve performance.

     Note: IBM reserves the right to change the calculation of the optimal send size.
3. QOPT File System Differences

   The value for st_atime will always be zero. The value for st_ctime will always be the creation date and time of the file or directory.

   The user, group, and other mode bits are always on for an object that exists on a volume not formatted in Universal Disk Format (UDF).

   fstat() on /QOPT will always return 2,147,483,647 for size fields.

   fstat() on optical volumes will return the volume capacity or 2,147,483,647, whichever is smaller.

   The file access time is not changed.
4. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)
5. QNetWare File System Differences

   The QNetWare file system does not fully support mode bits. See the Netware on iSeries topic for more information.
6. QFileSvr.400 File System Differences

   The value of st_vfs will always be 0 for remote objects accessed via QFileSvr.400.
7. This function will fail with the [EOVERFLOW] error if the specified file exists and its size is too large to be represented in the structure pointed to by *buffer* (the file is larger than 2GB minus 1 byte).
8. When you develop in C-based languages and this function is compiled with _LARGE_FILES defined, it will be mapped to **fstat64()**. Note that the type of the *buffer* parameter, `struct stat *`, also will be mapped to type `struct stat64 *`. See "stat64()—Get File Information (Large File Enabled)" on page 475 for more information on this structure.
9. If a descriptor for a pipe or socket is passed to this function, the value of st_vfs will be 0. Therefore, information about these objects' corresponding file system cannot be obtained using the QP0L_RETRIEVE_MOUNTED_FILE_SYSTEMS option of "Perform File System Operation (QP0LFLOP) API" on page 315.

Related Information

- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**sys/stat.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "fstat64()—Get File Information by Descriptor (Large File Enabled)" on page 99—Get File Information by Descriptor (Large File Enabled)
- "lstat()—Get File or Link Information" on page 162—Get File or Link Information

- “open()—Open File” on page 195—Open File
- socket()—Create Socket
- “stat()—Get File Information” on page 468—Get File Information
- “stat64()—Get File Information (Large File Enabled)” on page 475—Get File Information (Large File Enabled)
- » “Perform File System Operation (QP0LFLOP) API” on page 315—Perform file system operation«

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets status information:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <time.h>

main() {
  char fn[]="temp.file";
  struct stat info;
  int file_descriptor;

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    if (fstat(file_descriptor, &info) != 0)
      perror("fstat() error");
    else {
      puts("fstat() returned:");
      printf("  inode:   %d\n",   (int) info.st_ino);
      printf(" dev id:   %d\n",   (int) info.st_dev);
      printf("   mode:   %08x\n",       info.st_mode);
      printf("  links:   %d\n",         info.st_nlink);
      printf("    uid:   %d\n",   (int) info.st_uid);
      printf("    gid:   %d\n",   (int) info.st_gid);
    }
    close(file_descriptor);
    unlink(fn);
  }
}
```

Output: Note that the output may vary from system to system.

```
fstat() returned:
  inode:   3057
 dev id:   1
   mode:   03000080
  links:   1
    uid:   137
    gid:   500
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

fstat64()—Get File Information by Descriptor (Large File Enabled)

Syntax

```
#include <sys/stat.h>

int fstat64(int fildes, struct stat64 *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes."

The **fstat64()** function gets status information about the file specified by the open file descriptor *file_descriptor* and stores the information in the area of memory indicated by the *buf* argument. The status information is returned in a `stat64` structure, as defined in the **<sys/stat.h>** header file.

fstat64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:

- Using the **open64()** function (see "open64()—Open File (Large File Enabled)" on page 211).
- Using the **open()** function (see "open()—Open File" on page 195) with O_LARGEFILE set in the oflag parameter.

The elements of the `stat64` structure are described in "stat64()—Get File Information (Large File Enabled)" on page 475.

For additional information about parameters, authorities required, and error conditions, see "fstat()—Get File Information by Descriptor" on page 95.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **fstat64()** API and the `struct stat64` data type, you must compile the source with the _LARGE_FILE_API macro defined.
2. All of the usage notes for **fstat()** apply to **fstat64()**. See Usage Notes in the **fstat()** API.

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets status information:

```
#define _LARGE_FILE_API
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <time.h>

main() {
  char fn[]="temp.file";
  struct stat64 info;
  int file_descriptor;

  if ((file_descriptor = creat64(fn, S_IWUSR)) < 0)
    perror("creat64() error");
  else {
    if (ftruncate64(file_descriptor, 8589934662) != 0)
      perror("ftruncate64() error");
    else {
      if (fstat64(file_descriptor, &info) != 0)
        perror("fstat64() error");
      else {
        puts("fstat64() returned:");
        printf("  inode:   %d\n",   (int) info.st_ino);
        printf(" dev id:   %d\n",   (int) info.st_dev);
```

```
        printf("   mode:   %08x\n",       info.st_mode);
        printf("  links:   %d\n",         info.st_nlink);
        printf("    uid:   %d\n",   (int) info.st_uid);
        printf("    gid:   %d\n",   (int) info.st_gid);
        printf("   size:   %lld\n", (long long) info.st_size);
      }
    }
    close(file_descriptor);
    unlink(fn);
  }
}
```

Output: Note that the output may vary from system to system.

```
fstat64() returned:
  inode:   3057
 dev id:   1
   mode:   03000080
  links:   1
    uid:   137
    gid:   500
   size:   8589934662
```

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

fstatvfs()—Get File System Information by Descriptor

Syntax

```
#include <sys/statvfs.h>

int fstatvfs(int fildes, struct statvfs *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 103.

The **fstatvfs()** function gets status information about the file system that contains the file referenced by the open file descriptor *fildes*. The information is stored in the area of memory indicated by the *buf* argument. The status information is returned in a `statvfs` structure, as defined in the <**sys/statvfs.h**> header file.

Parameters

fildes (Input) The file descriptor of the file from which file system information is required.

buf (Output) A pointer to the area to which the information should be written.

The elements of the `statvfs` structure are described in "statvfs()—Get File System Information" on page 478. Signed fields of the `statvfs` structure that are not supported by the mounted file system will be set to -1.

Authorities

No authorization is required.

Return Value

0	**fstatvfs()** was successful. The information is returned in *buf*.
-1	**fstatvfs()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fstatvfs()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[EPERM (page 540)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	

Error condition	Additional information
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. ″Root″ (/) and QOpenSys File System Differences

   These file systems return the `f_flag` field with the `ST_NOSUID` flag bit turned off. However, support for the setuid/setgid semantics is limited to the ability to store and retrieve the `S_ISUID` and `S_ISGID` flags when these file systems are accessed from the Network File System server.
3. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

4. When you develop in C-based languages and an application is compiled with the `_LARGE_FILES` macro defined, the **fstatvfs()** API will be mapped to a call to the **fstatvfs64()**. Additionally, the `struct statvfs` data type will be mapped to a `struct statvfs64`.

Related Information

- The <**sys/statvfs.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "chown()—Change Owner and Group of File" on page 29—Change Owner and Group of File
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "dup()—Duplicate Open File Descriptor" on page 55—Duplicate Open File Descriptor
- "fclear()—Write (Binary Zeros) to Descriptor" on page 77—Write (Binary Zeros) to Descriptor
- "fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)" on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "fstatvfs64()—Get File System Information by Descriptor (64-Bit Enabled)" on page 105—Get File System Information by Descriptor (64-Bit Enabled)
- "link()—Create Link to File" on page 153—Create Link to File
- "open()—Open File" on page 195—Open File
- "read()—Read from Descriptor" on page 437—Read from Descriptor
- "statvfs()—Get File System Information" on page 478—Get File System Information
- "unlink()—Remove Link to File" on page 492—Remove Link to File
- "utime()—Set File Access and Modification Times" on page 497—Set File Access and Modification Times
- "write()—Write to Descriptor" on page 502—Write to Descriptor

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets status information about a file system:

```
#include <sys/statvfs.h>
#include <stdio.h>

main() {
  struct statvfs info;
  int fildes;

  if (-1 == (fildes = open("/",O_RDONLY)))
    perror("open() error");
  else if (-1 == fstatvfs(fildes, &info))
      perror("fstatvfs() error");
  else {
    puts("fstatvfs() returned the following information");
    puts("about the Root ('/') file system:");
    printf("  f_bsize    : %u\n", info.f_bsize);
    printf("  f_blocks   : %08X%08X\n",
                           *((int *)&info.f_blocks[0]),
                           *((int *)&info.f_blocks[4]));
    printf("  f_bfree    : %08X%08X\n",
                           *((int *)&info.f_bfree[0]),
                           *((int *)&info.f_bfree[4]));
    printf("  f_files    : %u\n", info.f_files);
    printf("  f_ffree    : %u\n", info.f_ffree);
    printf("  f_fsid     : %u\n", info.f_fsid);
    printf("  f_flag     : %X\n", info.f_flag);
```

```
    printf("  f_namemax  : %u\n", info.f_namemax);
    printf("  f_pathmax  : %u\n", info.f_pathmax);
    printf("  f_basetype : %s\n", info.f_basetype);
  }
}
```

Output: The following information will vary from file system to file system.

```
statvfs() returned the following information
about the Root ('/') file system:
  f_bsize    : 4096
  f_blocks   : 00000000002BF800
  f_bfree    : 0000000000091703
  f_files    : 4294967295
  f_ffree    : 4294967295
  f_fsid     : 0
  f_flag     : 1A
  f_namemax  : 255
  f_pathmax  : 4294967295
  f_basetype : "root" (/)
```

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

fstatvfs64()—Get File System Information by Descriptor (64-Bit Enabled)

Syntax

```
#include <sys/statvfs.h>

int fstatvfs64(int fildes, struct statvfs64 *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes."

The **fstatvfs64()** function gets status information about the file system that contains the file referred to by the open file descriptor *fildes*. The information is stored in the area of memory indicated by the *buf* argument. The status information is returned in a `statvfs64` structure, as defined in the <**sys/statvfs.h**> header file.

For details about parameters, authorities required, error conditions and examples, see "fstatvfs()—Get File System Information by Descriptor" on page 101. For details about the `struct statvfs64` structure, see "statvfs64()—Get File System Information (64-Bit Enabled)" on page 483.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **fstatvfs64()** API and the struct statvfs64 data type, you must compile the source with the _LARGE_FILE_API defined.
2. All of the usage notes for **fstatvfs()** apply to **fstatvfs64()**. See "Usage Notes" on page 103 in the **fstatvfs()** API.

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

fsync()—Synchronize Changes to File

Syntax

```
#include <unistd.h>

int fsync(int file_descriptor);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 107.

The **fsync()** function transfers all data for the file indicated by the open file descriptor *file_descriptor* to the storage device associated with *file_descriptor*. **fsync()** does not return until the transfer is complete, or until an error is detected.

Parameters

file_descriptor
(Input) The file descriptor of the file that is to have its modified data written to permanent storage.

Authorities

No authorization is required. Authorization is verified during **open()** or **creat()**.

Return Value

0 **fsync()** was successful.

-1 **fsync()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fsync()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EINVAL (page 540)]	For example, the file type is not valid for this operation.
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	

Error condition	Additional information
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     – "Root" (/)
     – QOpenSys
     – User-defined
     – QNTC
     – QSYS.LIB
     – Independent ASP QSYS.LIB

- QOPT
- Network File System
- QFileSvr.400

2. Using this function on a character special file will result in a return value of -1 and the errno global value set to EINVAL.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "fclear()—Write (Binary Zeros) to Descriptor" on page 77—Write (Binary Zeros) to Descriptor
- "fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)" on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)
- "open()—Open File" on page 195—Open File
- "write()—Write to Descriptor" on page 502—Write to Descriptor

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **fsync()**:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define mega_string_len 250000

main() {
  char *mega_string;
  int  file_descriptor;
  int  ret;
  char fn[]="fsync.file";

  if ((mega_string = (char*) malloc(mega_string_len)) == NULL)
    perror("malloc() error");
  else if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    memset(mega_string, 's', mega_string_len);
    if ((ret = write(file_descriptor,
                  mega_string, mega_string_len)) == -1)
      perror("write() error");
    else {
      printf("write() wrote %d bytes\n", ret);
      if (fsync(file_descriptor) != 0)
        perror("fsync() error");
      else if ((ret = write(file_descriptor,
                         mega_string, mega_string_len)) == -1)
        perror("write() error");
      else
        printf("write() wrote %d bytes\n", ret);
    }
    close(file_descriptor);
    unlink(fn);
  }
  free(mega_string);
}
```

Output:

```
write() wrote 250000 bytes
write() wrote 250000 bytes
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

ftruncate()—Truncate File

Syntax

```
#include <unistd.h>

int ftruncate(int file_descriptor, off_t length);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 111.

The **ftruncate()** function truncates the file indicated by the open file descriptor *file_descriptor* to the indicated *length*. *file_descriptor* must be a "regular file" that is open for writing. (A regular file is a stream file that can support positioning the file offset.) If the file size exceeds *length*, any extra data is discarded. If the file size is smaller than *length*, the file is extended and filled with binary zeros to the indicated length. (In the QSYS.LIB and independent ASP QSYS.LIB file systems blanks are used instead of zeros to pad records after a member is extended.) The **ftruncate()** function does not modify the current file offset for any open file descriptions associated with the file.

If **ftruncate()** completes successfully, it marks the change time and modification times of the file. Also, the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode are cleared. If **ftruncate()** is not successful, the file is unchanged.

If **ftruncate()** is used to truncate the file to 0 bytes and the file has a digital signature, the signature is deleted.

Parameters

file_descriptor
(Input) The file descriptor of the file.

length
(Input) The desired size of the file in bytes.

Authorities

No authorization is required. Authorization is verified during **open()** or **creat()**.

Return Value

0
ftruncate() was successful.

-1
ftruncate() was not successful. The *errno* global variable is set to indicate the error. If the file descriptor is not open for writing, **ftruncate** returns a [EBADF] error. If the file descriptor is a valid descriptor open for writing but is not a descriptor for a regular file, **ftruncate()** returns a [EINVAL] error.

Error Conditions

If **ftruncate()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	The QSYS.LIB or independent ASP QSYS.LIB file system cannot get exclusive access to the member to clear truncated data.
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFBIG (page 545)]	The size of the object would exceed the system allowed maximum size or the process soft file size limit. The file is a regular file and *length* is greater than 2GB minus 1 byte.
[EINTR (page 541)]	
[EINVAL (page 540)]	For example, *file_descriptor* does not refer to a regular file open for writing, or the specified length is not correct.
[EIO (page 540)]	
[EISDIR (page 544)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOCKED (page 545)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOSYS (page 544)]	
[ENOSYSRSC (page 545)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	

Error condition	Additional information
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. When you develop in C-based languages and this function is compiled with _LARGE_FILES defined, it will be mapped to **ftruncate64()**. Note also that the type of the *length* parameter will be remapped from off_t to off64_t.
3. For the Network File System, this function will fail with the [EFBIG] or the [EIO] error if the length specified is greater than the largest file size supported by the server.
4. Using this function on a character special file results in a return value of -1 and the errno global value set to EINVAL.
5. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   This function is not supported for save files and will fail with error code [ENOTSUP].
6. If the request exceeds the process soft file size limit, signal SIFXFSZ is issued.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "ftruncate64()—Truncate File (Large File Enabled)" on page 113—Truncate File (Large File Enabled)
- "open()—Open File" on page 195—Open File

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **ftruncate()**:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define string_len 1000

main() {
  char *mega_string;
  int  file_descriptor;
  int  ret;
  char fn[]="write.file";
  struct stat st;

  if ((mega_string = (char*) malloc(string_len)) == NULL)
    perror("malloc() error");
  else if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    memset(mega_string, '0', string_len);
    if ((ret = write(file_descriptor, mega_string, string_len)) == -1)
      perror("write() error");
    else {
      printf("write() wrote %d bytes\n", ret);
      fstat(file_descriptor, &st);
      printf("the file has %ld bytes\n", (long) st.st_size);
      if (ftruncate(file_descriptor, 1) != 0)
        perror("ftruncate() error");
      else {
        fstat(file_descriptor, &st);
        printf("the file has %ld bytes\n", (long) st.st_size);
      }
    }
    close(file_descriptor);
    unlink(fn);
  }
  free(mega_string);
}
```

Output:

```
write() wrote 1000 bytes
the file has 1000 bytes
the file has 1 bytes
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

ftruncate64()—Truncate File (Large File Enabled)

Syntax

```
#include <unistd.h>

int ftruncate64(int file_descriptor, off64_t length);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes."

The **ftruncate64()** function truncates the file indicated by the open file descriptor *file_descriptor* to the indicated *length*. *file_descriptor* must be a "regular file" that is open for writing. (A regular file is a stream file that can support positioning the file offset.) If the file size exceeds *length*, any extra data is discarded. If the file size is smaller than *length*, the file is extended and filled with binary zeros to the indicated length. (In the QSYS.LIB and independent ASP QSYS.LIB file systems, blanks are used instead of zeros to pad records after a member is extended.) The **ftruncate64()** function does not modify the current file offset for any open file descriptions associated with the file.

ftruncate64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:

- Using the **open64()** function (see "open64()—Open File (Large File Enabled)" on page 211).
- Using the **open()** function (see "open()—Open File" on page 195) with the O_LARGEFILE flag set in the oflag parameter.

If **ftruncate64()** completes successfully, it marks the change time and modification times of the file. If **ftruncate64()** is not successful, the file is unchanged.

For additional information about parameters, authorities, error conditions, and examples, see "ftruncate()—Truncate File" on page 109.

Usage Notes

1. For file systems that do support large files, this function will fail with the [EFBIG] error if the *length* specified is greater than 2GB minus 1 byte and O_LARGEFILE is not set in the oflag.
2. For file systems that do not support large files, this function will fail with the [EINVAL] error if the *length* specified is greater than 2GB minus 1 byte.
3. QFileSvr.400 File System Differences

   Although QFileSvr.400 does not support large files, it will return [EFBIG] if the *length* specified is greater than 2GB minus 1 byte.
4. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **ftruncate64()** API and the off64_t data type, you must compile the source with _LARGE_FILE_API defined.
5. All of the usage notes for **ftruncate()** apply to **ftruncate64()**. See "Usage Notes" on page 111 in the **ftruncate()** API.

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

getcwd()—Get Current Directory

Syntax

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Service Program Name: QP0LLIB2
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 116.

The **getcwd()** function determines the absolute path name of the current directory and stores it in *buf*. The components of the returned path name are not symbolic links.

The access time of each directory in the absolute path name of the current directory (excluding the current directory itself) is updated.

If *buf* is a NULL pointer, **getcwd()** returns a NULL pointer and the [EINVAL] error.

Parameters

buf (Output) A pointer to a buffer that will be used to hold the absolute path name of the current directory. The buffer must be large enough to contain the full pathname including the terminating NULL character. The current directory is returned in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See “QlgGetcwd()—Get Current Directory (using NLS-enabled path name)” on page 248 for a description and an example of supplying the *buf* in any CCSID.

size (Input) The number of bytes in the buffer *buf*.

Authorities

Note: Adopted authority is not used.

Authorization Required for getcwd()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the current directory	*RX	EACCES
Current directory	*X	EACCES
Note: QDLS File System Differences If the current directory is an immediate subdirectory of /QDLS (that is, at the next level below /QDLS in the directory hierarchy), the user must have *RX (*USE) authority to the directory. Otherwise, the QDLS authority requirements are the same as shown above.		

Return Value

value **getcwd()** was successful. The value returned is a pointer to *buf*.

NULL **getcwd()** was not successful. The *errno* global variable is set to indicate the error. After an error, the contents of *buf* are not defined.

Note: If *buf* is a NULL pointer, **getcwd()** returns a NULL pointer.

Error Conditions

If **getcwd()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EEXIST (page 543)]	
[EFAULT (page 541)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EMFILE (page 543)]	
[ENAMETOOLONG (page 544)]	
[ENFILE (page 543)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[ERANGE (page 540)]	The value of an argument is too small, or a result too large. For example, the **size** argument is too small. It is greater than zero but smaller than the length of the path name plus a NULL character.
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	

Error condition	**Additional information**
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
   - Where multiple threads exist in the job.
   - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. QOPT File System Differences

   If the directory exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the directory and preceding directories in the path name follows the rules described in Authorization Required for getcwd() (page 114). If the directory exists on a volume formatted in some other media format, no authorization checks are made on the directory or preceding directories. The volume authorization list is checked for *USE authority regardless of the volume media format.

Related Information

- The **<unistd.h>** file (see "Header Files for UNIX-Type Functions" on page 537)
- "chdir()—Change Current Directory" on page 19—Change Current Directory
- "QlgGetcwd()—Get Current Directory (using NLS-enabled path name)" on page 248—Get Current Directory

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines the current directory:

```
#include <unistd.h>
#include <stdio.h>
```

```
main()
{
  char cwd[1024];

  if (chdir("/tmp") != 0)
    perror("chdir() error()");
  else
  {
    if (getcwd(cwd, sizeof(cwd)) == NULL)
      perror("getcwd() error");
    else
      printf("current working directory is: %s\n", cwd);
  }
}
```

Output:

```
current working directory is: /tmp
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getegid()—Get Effective Group ID

Syntax

```
#include <unistd.h>

gid_t getegid(void);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: Yes

The **getegid()** function returns the effective group ID (GID) of the calling thread. The effective GID is the group ID under which the thread is currently running. The effective GID of a thread may change while the thread is running.

Parameters

None.

Authorities

No authorization is required.

Return Value

> 0 **getegid()** was successful. The value returned represents the effective *GID*.

>= 0 **getegid()** was successful. If there is no *GID*, the user ID has no group profile associated with it and returns 0. Otherwise, if there is a group profile, the API returns the *GID* of the group profile.

-1 **getegid()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **getegid()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	Internal object compressed. Try again.
[EDAMAGE (page 544)]	The user profile associated with the thread *GID* or an internal system object is damaged.
[ENOMEM (page 543)]	The user profile associated with the thread *GID* has exceeded its storage limit.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the effective GID.

```
#include <unistd.h>

main()
{
  gid_t ef_gid;

  if (-1 == (ef_gid = getegid(void)))
     perror("getegid() error.");
  else
     printf("The effective GID is: %u\n", ef_gid);

}
```

Output:

```
  The effective GID is: 75
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

geteuid()—Get Effective User ID

Syntax

```
#include <unistd.h>

uid_t geteuid(void);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: Yes

The **geteuid()** function returns the effective user ID (UID) of the calling thread. The effective UID is the user ID under which the thread is currently running. The effective UID of a thread may change while the thread is running.

Parameters

None.

Authorities

No authorization is required.

Return Value

0 or > 0
: **geteuid()** was successful. The value returned represents the effective UID.

-1
: **geteuid()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **geteuid()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	Internal object compressed. Try again.
[EDAMAGE (page 544)]	The user profile associated with the thread UID or an internal system object is damaged.
[ENOMEM (page 543)]	The user profile associated with the thread UID has exceeded its storage limit.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the effective UID.

```
#include <unistd.h>

main()
{
  uid_t ef_uid;

  if (-1 == (ef_uid = geteuid(void)))
     perror("geteuid() error.");
  else
     printf("The effective UID is: %u\n", ef_uid);

}
```

Output:

```
The effective UID is: 1957
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getgid()—Get Real Group ID

Syntax

```
#include <unistd.h>

gid_t getgid(void);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: Yes

The **getgid()** function returns the real group ID (GID) of the calling thread. The real GID is the group ID under which the thread was created.

Note:: When a user profile swap is done with the QWTSETP API prior to running the getgid() function, the GID for the current profile is returned.

Parameters

None.

Authorities

No authorization is required.

Return Value

> 0 **getgid()** was successful. The value returned represents the *GID*.

>= 0 **getgid()** was successful. If there is no *GID*, the user ID has no group profile associated with it and returns 0. Otherwise, if there is a group profile, the API returns the *GID* of the group profile.

-1 **getgid()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **getgid()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	Internal object compressed. Try again.
[EDAMAGE (page 544)]	The user profile associated with the thread *GID* or an internal system object is damaged.
[ENOMEM (page 543)]	The user profile associated with the thread *GID* has exceeded its storage limit.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the real GID.

```
#include <unistd.h>

main()
{
  gid_t gid;

  if (-1 == (gid = getgid(void)))
     perror("getgid() error.");
  else
     printf("The real GID is: %u\n", gid);

}
```

Output:

```
  The real GID is: 75
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

getgrgid()—Get Group Information Using Group ID

Syntax

```
#include <grp.h>

struct group *getgrgid(gid_t gid);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: No

The **getgrgid()** function returns a pointer to an object of type `struct group` containing an entry from the user database with a matching *GID*.

Parameters

gid (Input) Group ID.

Authorities

*READ authority is required to the user profile associated with the *gid*. If the user does not have *READ authority, only the name of the group and the group ID values are returned.

Return Value

*struct group **
: **getgrgid()** was successful. The return value points to static data of the format `struct group`, which is defined in the **grp.h** header file. This storage is overwritten on each call to this function. This static storage area is also used by the **getgrnam()** function. The `struct group` has the following elements:

char *	gr_name	Name of the group
gid_t	gr_gid	Group ID
char **	gr_mem	A null-terminated list of pointers to the individual member profile names. If the group profile does not have any members or if the caller does not have *READ authority to the group profile, the list will be empty.

NULL pointer
: **getgrgid** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **getgrgid()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	**Additional information**
[EAGAIN (page 541)]	The user profile associated with the *GID* is currently locked by another process.
[EC2]	Detected pointer that is not valid.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with the *GID* was not found.
[ENOMEM (page 543)]	The user profile associated with the *GID* has exceeded its storage limit.

Error condition	Additional information
[ENOSPC (page 541)]	Machine storage limit exceeded.

Related Information

- The <**grp.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "getgrgid_r()—Get Group Information Using Group ID"—Get Group Information Using Group ID

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the group information for the gid of 91. The group name is GROUP1. There are two group members, CLIFF and PATRICK.

```
#include <grp.h>
#include <stdio.h>

main()
{
  struct group *grp;
  short int    lp;

  if (NULL == (grp = getgrgid(91)))
     perror("getgrgid() error.");
  else
  {
     printf("The group name is: %s\n", grp->gr_name);
     printf("The gid        is: %u\n", grp->gr_gid);
     for (lp = 1; NULL != *(grp->gr_mem); lp++, (grp->gr_mem)++)
        printf("Group member %d is: %s\n", lp, *(grp->gr_mem));
  }

}
```

Output:

```
The group name is: GROUP1
The gid        is: 91
Group member 1 is: CLIFF
Group member 2 is: PATRICK
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getgrgid_r()—Get Group Information Using Group ID

Syntax

```
#include <sys/types.h>
#include <grp.h>

int getgrgid_r(gid_t gid, struct group *grp,
char *buffer, size_t bufsize, struct group
**result);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: Yes

The **getgrgid_r()** function updates the group structure pointed to by *grp* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *GID*.

Parameters

gid (Input) Group ID.

grp (Input) A pointer to a group structure.

buffer (Input) A pointer to a buffer from which memory is allocated to hold storage areas referenced by the group structure *grp*.

bufsize
(Input) The size of *buffer* in bytes.

result (Input) A pointer to a location in which a pointer to the updated group structure is stored. If an error occurs or if the requested entry cannot be found, a NULL pointer is stored in this location.

The `struct group`, which is defined in the **grp.h** header file, has the following elements:

char *	gr_name	Name of the group
gid_t	gr_gid	Group ID
char **	gr_mem	A null-terminated list of pointers to the individual member profile names. If the group profile does not have any members or if the caller does not have *READ authority to the group profile, the list will be empty.

Authorities

*READ authority is required to the user profile associated with the *gid*. If the user does not have *READ authority, only the name of the group and the group ID values are returned.

Return Value

0 **getgrgid_r** was successful.

Any other value
Failure: The return value contains an error number indicating the error.

Error Conditions

If **getgrgid_r()** is not successful, the return value usually indicates one of the following errors. Under some conditions, the value could indicate an error other than those listed here.

Error condition	**Additional information**
[EAGAIN (page 541)]	The user profile associated with the *GID* is currently locked by another process.
[EC2]	Detected pointer that is not valid.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with the *GID* was not found.
[ENOMEM (page 543)]	The user profile associated with the *GID* has exceeded its storage limit.
[ENOSPC (page 541)]	Machine storage limit exceeded.
[ERANGE (page 540)]	Insufficient storage was supplied by *buffer* and *bufsize* to contain the data to be referenced by the resulting group structure.

Related Information

- The <**grp.h**> file "Header Files for UNIX-Type Functions" on page 537(see)

- "getgrgid()—Get Group Information Using Group ID" on page 121—Get Group Information Using Group ID

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the group information for the gid of 91. The group name is GROUP1. There are two group members, CLIFF and PATRICK.

```
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
#include <errno.h>

main()
{ short int lp;
  struct group grp;
  struct group * grpptr=&grp;
  struct group * tempGrpPtr;
  char grpbuffer[200];
  int  grplinelen = sizeof(grpbuffer);

  if ((getgrgid_r(91,grpptr,grpbuffer,grplinelen,&tempGrpPtr))!=0)
     perror("getgrgid_r() error.");
  else
  {
     printf("\nThe group name is: %s\n", grp.gr_name);
     printf("The gid        is: %u\n", grp.gr_gid);
     for (lp = 1; NULL != *(grp.gr_mem); lp++, (grp.gr_mem)++)
        printf("Group Member %d is: %s\n", lp, *(grp.gr_mem));
  }

}
```

Output:

```
  The group name is: GROUP1
  The gid        is: 91
  Group member 1 is: CLIFF
  Group member 2 is: PATRICK
```

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

---

getgrgid_r_ts64()—Get Group Information Using Group ID

Syntax

```
#include <sys/types.h>
#include <grp.h>

int getgrgid_r_ts64(
       gid_t gid,
       struct group * __ptr64 grp,
       char * __prt64 buffer,
       size_t bufsize,
       struct group * __ptr64 * __ptr64 result);
```

Service Program Name: QSYPAPI64
Default Public Authority: *USE
Threadsafe: Yes

The **getgrgid_r_ts64()** function updates the group structure pointed to by *grp* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *GID*. **getgrgid_r_ts64()** differs from **getgrgid_r()** in that it accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, related information, usage notes, and an example for the **getgrgid_r()** API, see "getgrgid_r()—Get Group Information Using Group ID" on page 122—Get Group Information Using Group ID.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

---

getgrnam()—Get Group Information Using Group Name

Syntax

```
#include <grp.h>

struct group  *getgrnam(const char *name);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: No

The **getgrnam()** function returns a pointer to an object of type `struct group` containing an entry from the user database with a matching *name*.

Parameters

name (Input) A pointer to a group profile name.

Authorities

*READ authority is required to the user profile associated with the *name*. If the user does not have *READ authority, only the name of the group and the group ID values are returned.

Return Value

*struct group **

getgrnam() was successful. The return value points to static data of the format `struct group`, which is defined in the **grp.h** header file. This storage is overwritten on each call to this function. This static storage area is also used by the **getgrgid()** function. The `struct group` has the following elements:

char *	gr_name	Name of the group
gid_t	gr_gid	Group ID
char **	gr_mem	A null-terminated list of pointers to the individual member profile names. If the group profile does not have any members or if the caller does not have *READ authority to the group profile, the list will be empty.

NULL pointer

getgrnam was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **getgrnam()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	The user profile associated with the *name* is currently locked by another process.
[EC2]	Detected pointer that is not valid.
[EDAMAGE (page 544)]	The user profile associated with the group name or an internal system object is damaged.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with the *name* was not found or the profile name specified is not a group profile.
[EUNKNOWN (page 544)]	Unknown system state. Check the job log for a CPF9872 message.

Related Information

- The <**grp.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "getgrnam_r()—Get Group Information Using Group Name" on page 127—Get Group Information Using Group Name

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the group information for the group GROUP1. The gid is 91. There are two group members, CLIFF and PATRICK.

```
#include <grp.h>
#include <stdio.h>

main()
{
  struct group *grp;
  short int    lp;

  if (NULL == (grp = getgrnam("GROUP1")))
     perror("getgrnam() error.");
  else
  {
     printf("The group name is: %s\n", grp->gr_name);
     printf("The gid        is: %u\n", grp->gr_gid);
     for (lp = 1; NULL != *(grp->gr_mem); lp++, (grp->gr_mem)++)
        printf("Group member %d is: %s\n", lp, *(grp->gr_mem));
  }

}
```

Output:

```
  The group name is: GROUP1
  The gid        is: 91
  Group member 1 is: CLIFF
  Group member 2 is: PATRICK
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

getgrnam_r()—Get Group Information Using Group Name

Syntax

```
#include <sys/types.h>
#include <grp.h>

int getgrnam_r(const char *name, struct group *grp,
char *buffer, size_t bufsize, struct group
**result);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: Yes

The **getgrnam_r()** function updates the group structure pointed to by *grp* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with matching *name*.

Parameters

name (Input) A pointer to a group profile name.

grp (Input) A pointer to a group structure.

buffer (Input) A pointer to a buffer from which memory is allocated to hold storage areas referenced by the group structure *grp*.

bufsize
(Input) The size of *buffer* in bytes.

result (Input) A pointer to a location in which a pointer to the updated group structure is stored. If an error occurs or the requested entry cannot be found, a NULL pointer is stored in this location.

The `struct group`, which is defined in the **grp.h** header file, has the following elements:

char *	gr_name	Name of the group
gid_t	gr_gid	Group ID
char **	gr_mem	A null-terminated list of pointers to the individual member profile names. If the group profile does not have any members or if the caller does not have *READ authority to the group profile, the list will be empty.

Authorities

*READ authority is required to the user profile associated with the *name*. If the user does not have *READ authority, only the name of the group and the group ID values are returned.

Return Value

0 **getgrnam_r** was successful.

Any other value
Failure: The return value contains an error number indicating the error.

Error Conditions

If **getgrnam_r()** is not successful, the return value usually indicates one of the following errors. Under some conditions, the value could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	The user profile associated with the *name* is currently locked by another process.
[EC2]	Detected pointer that is not valid.
[EDAMAGE (page 544)]	The user profile associated with the group name or an internal system object is damaged.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with the *name* was not found or the profile name specified is not a group profile.
[ERANGE (page 540)]	Insufficient storage was supplied by *buffer* and *bufsize* to contain the data to be referenced by the resulting group structure.
[EUNKNOWN (page 544)]	Unknown system state. Check the job log for a CPF9872 message.

Related Information

- The <**grp.h**> file (see)
- "getgrnam_r()—Get Group Information Using Group Name" on page 127—Get Group Information Using Group Name

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the group information for the group GROUP1. The gid is 91. There are two group members, CLIFF and PATRICK.

```
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
#include <errno.h>

main()
{ short int lp;
  struct group grp;
  struct group * grpptr=&grp;
  struct group * tempGrpPtr;
  char grpbuffer[200];
  int  grplinelen = sizeof(grpbuffer);

  if ((getgrnam_r("GROUP1",grpptr,grpbuffer,grplinelen,&tempGrpPtr))!=0)
     perror("getgrnam_r() error.");
  else
  {
     printf("\nThe group name is: %s\n", grp.gr_name);
     printf("The gid         is: %u\n", grp.gr_gid);
     for (lp = 1; NULL != *(grp.gr_mem); lp++, (grp.gr_mem)++)
        printf("Group Member %d is: %s\n", lp, *(grp.gr_mem));
  }

}
```

Output:

```
  The group name is: GROUP1
  The gid        is: 91
  Group member 1 is: CLIFF
  Group member 2 is: PATRICK
```

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

getgrnam_r_ts64()—Get Group Information Using Group Name

Syntax

```
#include <sys/types.h>
#include <grp.h>

int getgrnam_r_ts64(
       const char * __ptr64 name,
       struct group * __ptr64 grp,
       char * __ptr64 buffer,
       size_t bufsize,
       struct group * __ptr64 * __ptr64 result);
```

Service Program Name: QSYPAPI64
Default Public Authority: *USE
Threadsafe: Yes

The **getgrnam_r_ts64()** function updates the group structure pointed to by *grp* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *name*. **getgrnam_r_ts64()** differs from **getgrnam_r()** in that it accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, related information, usage notes, and an example for the **getgrnam_r()** API, see "getgrnam_r()—Get Group Information Using Group Name" on page 127—Get Group Information Using Group Name.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

getgroups()—Get Group IDs

Syntax

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[])
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: No

If the *gidsetsize* argument is zero, **getgroups()** returns the number of group IDs associated with the calling thread without modifying the array pointed to by the *grouplist* argument. The number of group IDs includes the effective group ID and the supplementary group IDs. Otherwise, **getgroups()** fills in the array *grouplist* with the effective group ID and supplementary group IDs of the calling thread and returns the actual number of group IDs stored. The values of array entries with indexes larger than or equal to the returned value are undefined.

Parameters

gidsetsize
: (Input) The number of elements in the supplied array *grouplist*.

grouplist
: (Output) The effective group ID and supplementary group IDs. The first element in *grouplist* is the effective group ID.

Authorities

No authorization is required.

Return Value

0 or > 0	**getgroups()** was successful. If the *gidsetsize* argument is 0, the number of group IDs is returned. This number includes the effective group ID and supplementary group IDs. If *gidsetsize* is greater than 0, the array *grouplist* is filled with the effective group ID and supplementary group IDs of the calling thread and the return value represents the actual number of group IDs stored.
-1	**getgroups()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **getgroups()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EINVAL (page 540)]	The *gidsetsize* argument is not equal to zero and is less than the number of group IDs.

Usage Notes

This function can be used in two different ways. First, if called with *gidsetsize* equal to 0, it is used to return the number of groups associated with a thread. Second, if called with *gidsetsize* not equal to 0, it is used to return a list of the GIDs representing the effective and supplementary groups associated with a thread. In this case, the *gidsetsize* argument represents how much space is available in the *grouplist* argument.

The calling routine can choose to call this function with *gidsetsize* equal to 0 to determine how much space to allocate for a second call to this function. The second call returns the values. The following is an example of this method:

```
int numgroups;
gid_t *grouplist;

numgroups = getgroups(0,NULL);
grouplist = (gid_t *) calloc( numgroups, sizeof(gid_t) );
if (getgroups( numgroups, grouplist) != -1) {
        .
        .
}
```

Alternatively, the calling routine can choose to create enough space for NGROUPS_MAX entries to ensure enough space is available for the maximum possible number of entries that may be returned. This introduces the possibility of wasted space. The following is an example of this method:

```
int numgroups;
gid_t grouplist[ NGROUPS_MAX ];

if ( getgroups( NGROUPS_MAX, grouplist ) != -1 ) {
        .
        .
}
```

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

getpwnam()—Get User Information for User Name

Syntax

```
#include <pwd.h>

struct passwd *getpwnam(const char *name);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: No

The **getpwnam()** function returns a pointer to an object of type `struct passwd` containing an entry from the user database with a matching *name*.

Parameters

name (Input) A pointer to a user profile name.

Authorities

*READ authority is required to the user profile associated with the *name*. If the user does not have *READ authority, only the user name, user ID, and group ID values are returned.

Note: Adopted authority is not used.

Return Value

*struct passwd **
getpwnam() was successful. The return value points to static data of the format `struct passwd`, which is defined in the **pwd.h** header file. This storage is overwritten on each call to this function. This static storage area is also used by the **getpwuid()** function. The `struct passwd` has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID
uid_t	pw_gid	Group ID of the user's first group. If the user does not have a first group, the gid value will be set to 0.
char *	pw_dir	Initial working directory. If the user does not have *READ authority to the user profile, the pw_dir pointer will be set to NULL.
char *	pw_shell	Initial user program. If the user does not have *READ authority to the user profile, the pw_shell pointer will be set to NULL.

NULL pointer
getpwnam() was not successful. The *errno* global variable is set to indicate the error.

See "QlgGetpwnam()—Get User Information for User Name (using NLS-enabled path name)" on page 252 for a description and an example where the path name is returned in any CCSID.

Error Conditions

If **getpwnam()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	The user profile associated with the *name* is currently locked by another process.
[EC2]	Detected pointer that is not valid.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with the *name* was not found.
[ENOMEM (page 543)]	The user profile associated with the *UID* has exceeded its storage limit or is unable to allocate memory.
[EUNKNOWN (page 544)]	Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The initial working directory is returned in the CCSID value of the job.

Related Information

- The <**pwd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "getpwnam_r()—Get User Information for User Name" on page 133—Get User Information for User Name
- "QlgGetpwnam()—Get User Information for User Name (using NLS-enabled path name)" on page 252—Get User Information for User Name (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the user database information for the user name of MYUSER. The UID is 22. The gid of MYUSER's first group is 1012. The initial directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <pwd.h>

main()
{
  struct passwd *pd;

  if (NULL == (pd = getpwnam("MYUSER")))
     perror("getpwnam() error.");
  else
  {
     printf("The user name is: %s\n", pd->pw_name);
     printf("The user id   is: %u\n", pd->pw_uid);
     printf("The group id  is: %u\n", pd->pw_gid);
     printf("The initial directory is:    %s\n", pd->pw_dir);
     printf("The initial user program is: %s\n", pd->pw_shell);
  }

}
```

Output:

```
  The user name is: MYUSER
  The user id   is: 22
  The group id  is: 1012
  The initial directory is:    /home/MYUSER
  The initial user program is: *LIBL/QCMD
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getpwnam_r()—Get User Information for User Name

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int getpwnam_r(const char *name, struct passwd
*pwd, char *buffer, size_t bufsize,
struct passwd **result);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: Yes

The **getpwnam_r()** function updates the *passwd* structure pointed to by *pwd* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *name*.

Parameters

name (Input) A pointer to a user profile name.

pwd (Input) A pointer to a passwd structure.

buffer (Input) A pointer to a buffer from which memory is allocated to hold storage areas referenced by the structure *pwd*.

bufsize
(Input) The size of *buffer* in bytes.

result (Input) A pointer to a location in which a pointer to the updated passwd structure is stored. If an error occurs or if the requested entry cannot be found, a NULL pointer is stored in this location.

The `struct passwd`, which is defined in the **pwd.h** header file, has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID
uid_t	pw_gid	Group ID of the user's first group. If the user does not have a first group, the GID value will be set to 0.
char *	pw_dir	Initial working directory. If the user does not have *READ authority to the user profile, the pw_dir pointer will be set to NULL.
char *	pw_shell	Initial user program. If the user does not have *READ authority to the user profile, the pw_shell will be set to NULL.

See "QlgGetpwnam_r()—Get User Information for User Name (using NLS-enabled path name)" on page 254 for a description and an example where the path name is returned in any CCSID. Go to _r version

Authorities

*READ authority is required to the user profile associated with the *name*. If the user does not have *READ authority, only the user name, user ID, and group ID values are returned.

Note: Adopted authority is not used.

Return Value

0 **getpwnam_r** was successful.

Any other value
Failure: The return value contains an error number indicating the error.

Error Conditions

If **getpwnam_r()** is not successful, the return value usually indicates one of the following errors. Under some conditions, the value could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	The user profile associated with the *name* is currently locked by another process.
[EC2]	Detected pointer that is not valid.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with the *name* was not found.
[ENOMEM (page 543)]	The user profile associated with the *UID* has exceeded its storage limit or is unable to allocate memory.
[ERANGE (page 540)]	Insufficient storage was supplied through *buffer* and *bufsize* to contain the data to be referenced by the resulting group structure.
[EUNKNOWN (page 544)]	Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The initial working directory is returned in the CCSID value of the job.

Related Information

- The <**pwd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "getpwnam()—Get User Information for User Name" on page 131—Get User Information for User Name

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the user database information for the user name of MYUSER. The UID is 22. The GID of MYUSER's first group is 1012. The initial directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <errno.h>

main()
{
  struct passwd pd;
  struct passwd* pwdptr=&pd;
  struct passwd* tempPwdPtr;
  char pwdbuffer[200];
  int  pwdlinelen = sizeof(pwdbuffer);

  if ((getpwnam_r("MYUSER",pwdptr,pwdbuffer,pwdlinelen,&tempPwdPtr))!=0)
     perror("getpwnam_r() error.");
  else
  {
     printf("\nThe user name is: %s\n", pd.pw_name);
     printf("The user id   is: %u\n", pd.pw_uid);
     printf("The group id  is: %u\n", pd.pw_gid);
```

```
    printf("The initial directory is:    %s\n", pd.pw_dir);
    printf("The initial user program is: %s\n", pd.pw_shell);
  }

}
```

Output:

```
The user name is: MYUSER
The user ID   is: 22
The group ID  is: 1012
The initial directory is:    /home/MYUSER
The initial user program is: *LIBL/QCMD
```

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

getpwnam_r_ts64()—Get User Information for User Name

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int getpwnam_r_ts64(
       const char * __ptr64 name,
       struct passwd * __ptr64 pwd,
       char * __ptr64 buffer,
       size_t bufsize,
       struct passwd * __ptr64 * __ptr64 result);
```

Service Program Name: QSYPAPI64
Default Public Authority: *USE
Threadsafe: Yes

The **getpwnam_r_ts64()** function updates the *passwd* structure pointed to by *pwd* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *name*. **getpwnam_r_ts64()** differs from **getpwnam_r()** in that it accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, related information, usage notes, and an example for the **getpwnam_r()** API, see "getpwnam_r()—Get User Information for User Name" on page 133—Get User Information for User Name.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

getpwuid()—Get User Information for User ID

Syntax

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: No

The **getpwuid()** function returns a pointer to an object of type `struct passwd` containing an entry from the user database with a matching *UID*.

Parameters

uid (Input) User ID.

Authorities

*READ authority is required to the user profile associated with the *UID*. If the user does not have *READ authority, only the user name, user ID, and group ID values are returned.

Note: Adopted authority is not used.

Return Value

*struct passwd **

getpwuid() was successful. The return value points to static data of the format `struct passwd`, which is defined in the **pwd.h** header file. This storage is overwritten on each call to this function. This static storage area is also used by the **getpwnam()** function. The `struct passwd` has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID
uid_t	pw_gid	Group ID of the user's first group. If the user does not have a first group, the gid value will be set to 0.
char *	pw_dir	Initial working directory. If the user does not have *READ authority to the user profile, the pw_dir pointer will be set to NULL.
char *	pw_shell	Initial user program. If the user does not have *READ authority to the user profile, the pw_shell pointer will be set to NULL.

NULL pointer

getpwuid() was not successful. The *errno* global variable is set to indicate the error.

See "QlgGetpwuid()—Get User Information for User ID (using NLS-enabled path name)" on page 256 for a description and an example where the path name is returned in any CCSID.

Error Conditions

If **getpwuid()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	**Additional information**
[EAGAIN (page 541)]	The user profile associated with the *UID* is currently locked by another process.
[EC2]	Detected pointer that is not valid.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with *UID* was not found.
[ENOMEM (page 543)]	The user profile associated with the *UID* has exceeded its storage limit or is unable to allocate memory.
[ENOSPC (page 541)]	Machine storage limit exceeded.
[EUNKNOWN (page 544)]	Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The initial working directory is returned in the CCSID value of the job.

Related Information

- The <**pwd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "getpwuid_r()—Get User Information for User ID"—Get User Information for User ID
- "QlgGetpwuid()—Get User Information for User ID (using NLS-enabled path name)" on page 256—Get User Information for User ID (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the user database information for the UID of 22. The user name is MYUSER. The gid of MYUSER's first group is 1012. The initial directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <pwd.h>

main()
{
  struct passwd *pd;

  if (NULL == (pd = getpwuid(22)))
     perror("getpwuid() error.");
  else
  {
     printf("The user name is: %s\n", pd->pw_name);
     printf("The user id   is: %u\n", pd->pw_uid);
     printf("The group id  is: %u\n", pd->pw_gid);
     printf("The initial directory is:    %s\n", pd->pw_dir);
     printf("The initial user program is: %s\n", pd->pw_shell);
  }

}
```

Output:

```
  The user name is: MYUSER
  The user id   is: 22
  The group id  is: 1012
  The initial directory is:    /home/MYUSER
  The initial user program is: *LIBL/QCMD
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getpwuid_r()—Get User Information for User ID

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int getpwuid_r(uid_t uid, struct passwd *pwd,
char *buffer, size_t bufsize, struct passwd
**result);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: Yes

The **getpwuid_r()** function updates the *passwd* structure pointed to by *pwd* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *uid*.

Parameters

uid (Input) User ID.

pwd (Input) A pointer to a struct passwd.

buffer (Input) A pointer to a buffer from which memory is allocated to hold storage areas referenced by the structure passwd.

bufsize
(Input) The size of *buffer* in bytes.

result (Input) A pointer to a location in which a pointer to the updated passwd structure is stored. If an error occurs or if the requested entry cannot be found, a NULL pointer is stored in this location.

The `struct passwd`, which is defined in the **pwd.h** header file, has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID
uid_t	pw_gid	Group ID of the user's first group. If the user does not have a first group, the GID value will be set to 0.
char *	pw_dir	Initial working directory. If the user does not have *READ authority to the user profile, the pw_dir pointer will be set to NULL.
char *	pw_shell	Initial user program. If the user does not have *READ authority to the user profile, the pw_shell pointer will be set to NULL.

See "QlgGetpwuid_r()—Get User Information for User ID (using NLS-enabled path name)" on page 258 for a description and an example where the path name is returned in any CCSID.

Authorities

*READ authority is required to the user profile associated with the *UID*. If the user does not have *READ authority, only the user name, user ID, and group ID values are returned.

Note: Adopted authority is not used.

Return Value

0 **getpwuid_r()** was successful.

Any other value
Failure: The return value contains an error number indicating the error.

Error Conditions

If **getpwuid_r()** is not successful, the error value usually indicates one of the following errors. Under some conditions, the value could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	The user profile associated with the *UID* is currently locked by another process.
[EC2]	Detected pointer that is not valid.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with the *UID* was not found.

Error condition	Additional information
[ENOMEM (page 543)]	The user profile associated with the *UID* has exceeded its storage limit or is unable to allocate memory.
[ENOSPC (page 541)]	Machine storage limit exceeded.
[ERANGE (page 540)]	Insufficient storage was supplied through *buffer* and *bufsize* to contain the data to be referenced by the resulting group structure.
[EUNKNOWN (page 544)]	Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The initial working directory is returned in the CCSID value of the job.

Related Information

- The <**pwd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "getpwuid()—Get User Information for User ID" on page 135—Get User Information for User ID

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the user database information for the UID of 22. The user name is MYUSER. The GID of MYUSER's first group is 1012. The initial directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <errno.h>

main()
{
  struct passwd pd;
  struct passwd* pwdptr=&pd;
  struct passwd* tempPwdPtr;
  char pwdbuffer[200];
  int  pwdlinelen = sizeof(pwdbuffer);

  if ((getpwuid_r(22,pwdptr,pwdbuffer,pwdlinelen,&tempPwdPtr))!=0)
     perror("getpwuid_r() error.");
  else
  {
     printf("\nThe user name is: %s\n", pd.pw_name);
     printf("The user id   is: %u\n", pd.pw_uid);
     printf("The group id  is: %u\n", pd.pw_gid);
     printf("The initial directory is:    %s\n", pd.pw_dir);
     printf("The initial user program is: %s\n", pd.pw_shell);
  }

}
```

Output:

```
  The user name is: MYUSER
  The user ID   is: 22
  The group ID  is: 1012
  The initial directory is:    /home/MYUSER
  The initial user program is: *LIBL/QCMD
```

API introduced: V4R4

getpwuid_r_ts64()—Get User Information for User ID

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int getpwuid_r_ts64(
       uid_t uid,
       struct passwd * __ptr64 pwd,
       char * __ptr64 buffer,
       size_t bufsize,
       struct passwd * __ptr64 * __ptr64 result);
```

Service Program Name: QSYPAPI64
Default Public Authority: *USE
Threadsafe: Yes

The **getpwuid_r_ts64()** function updates the *passwd* structure pointed to by *pwd* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *UID*. **getpwuid_r_ts64()** differs from **getpwuid_r()** in that it accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, related information, usage notes, and an example for the **getpwuid_r()** API, see "getpwuid_r()—Get User Information for User ID" on page 137—Get User Information for User ID.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

getuid()—Get Real User ID

Syntax

```
#include <unistd.h>

uid_t getuid(void);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: Yes

The **getuid()** function returns the real user ID (UID) of the calling thread. The real UID is the user ID under which the thread was created.

Note: When a user profile swap is done with the QWTSETP API prior to running the getuid() function, the UID for the current profile is returned.

Parameters

None.

Authorities

No authorization is required.

Return Value

0 or > 0
: **getuid()** was successful. The value returned represents the *UID*.

-1
: **getuid()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **getuid()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	Internal object compressed. Try again.
[EDAMAGE (page 544)]	The user profile associated with the thread *UID* or an internal system object is damaged.
[ENOMEM (page 543)]	The user profile associated with the thread *UID* has exceeded its storage limit.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the real UID.

```
#include <unistd.h>

main()
{
  uid_t uid;

  if (-1 == (uid = getuid(void)))
     perror("getuid() error.");
  else
     printf("The real UID is: %u\n", uid);

}
```

Output:

```
  The real UID is: 1957
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

ioctl()—Perform I/O Control Request

Syntax

```
#include <sys/types.h>
#include <sys/ioctl.h>

int ioctl(int descriptor,
          unsigned long request,
   ...);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 148.

The **ioctl()** function performs control functions (requests) on a descriptor.

Parameters

descriptor
(Input) The descriptor on which the control request is to be performed.

request
(Input) The request that is to be performed on the *descriptor*.

... (Input) A variable number of optional parameters that are dependent on the request.

The *ioctl()* requests that are supported are:

FIOASYNC Set or clear the flag that allows the receipt of asynchronous I/O signals (SIGIO).

The third parameter represents a pointer to an integer flag. A nonzero value sets the socket to generate SIGIO signals, while a zero value sets the socket to not generate SIGIO signals. Note that before the SIGIO signals can be delivered, you must use either the FIOSETOWN or SIOCSPGRP *ioctl()* request, or the F_SETOWN *fcntl()* command to set a process ID or a process group ID to indicate what process or group of processes will receive the signal. Once conditioned to send SIGIO signals, a socket will generate SIGIO signals whenever certain significant conditions change on the socket. For example, SIGIO will be generated when normal data arrives on the socket, when out-of-band data arrives on the socket (in addition to the SIGURG signal), when an error occurs on the socket, or when end-of-file is received on the socket. It is also generated when a connection request is received on the socket (if it is a socket on which the *listen()* verb has been done). Also note that a socket can be set to generate the SIGIO signal by using the *fcntl()* command F_SETFL with a flag value specifying FASYNC.

FIOCCSID Return the coded character set ID (CCSID) associated with the open instance represented by the descriptor and the CCSID associated with the object. The third parameter represents a pointer to the structure Qp0lFIOCCSID, which is defined in <sys/ioctl.h>. This information may be necessary to correctly manipulate data read from or written to a file opened in another process.

If the open instance represented by the descriptor is in binary mode (the open() did not specify the O_TEXTDATA open flag), the open instance CCSID returned is equal to the object CCSID returned.

FIOGETOWN Get the process ID or process group ID that is to receive the SIGIO and SIGURG signals.

The third parameter represents a pointer to a signed integer that will contain the process ID or the process group ID to which the socket is currently sending asynchronous signals such as SIGURG. A process ID is returned as a positive integer, and a process group ID is specified as a negative integer. A 0 value returned indicates that no asynchronous signals can be generated by the socket. A positive or a negative value indicates that the socket has been set to generate SIGURG signals.

FIONBIO Set or clear the nonblocking I/O flag (O_NONBLOCK oflag). The third parameter represents a pointer to an integer flag. A nonzero value sets the nonblocking I/O flag for the descriptor; a zero value clears the flag.

FIONREAD Return the number of bytes available to be read. The third parameter represents a pointer to an integer that is set to the number of bytes available to be read.

FIOSETOWN

Set the process ID or process group ID that is to receive the SIGIO and SIGURG signals.

The third parameter represents a pointer to a signed integer that contains the process ID or the process group ID to which the socket should send asynchronous signals such as SIGURG. A process ID is specified as a positive integer, and a process group ID is specified as a negative integer. Specifying a 0 value resets the socket such that no asynchronous signals are delivered. Specifying a process ID or a process group ID requests that sockets begin sending the SIGURG signal to the specified ID when out-of-band data arrives on the socket.

SIOCADDRT

Add an entry to the interface routing table. Valid for sockets with address family of AF_INET.

The third parameter represents a pointer to the structure **rtentry**, which is defined in **<net/route.h>**:

```
struct rtentry [
   struct sockaddr rt_dst;
   struct sockaddr rt_mask;
   struct sockaddr rt_gateway;
   int rt_mtu;
   u_short rt_flags;
   u_short rt_refcnt;
   u_char rt_protocol;
   u_char rt_TOS;
   char rt_if[IFNAMSIZ];
];
```

The *rt_dst*, *rt_mask*, and *rt_gateway* fields are the route destination address, route address mask, and gateway address, respectively. *rt_mtu* is the maximum transfer unit associated with the route. *rt_flags* contains flags that give some information about a route (for example, whether the route was created dynamically, whether the route is usable, type of route, and so on). *rt_refcnt* indicates the number of references that exist to the route entry. *rt_protocol* indicates how the route entry was generated (for example, configuration, ICMP redirect, and so on). *rt_tos* is the type of service associated with the route. *rt_if* is a NULL-terminated string that represents the interface IP address in dotted decimal format that is associated with the route.

To add a route, the following fields must be set:

- *rt_dst*
- *rt_mask*
- *rt_gateway*
- *rt_tos*
- *rt_protocol*
- *rt_mtu* (Setting the *rt_mtu* value to zero essentially means use the MTU from the associated line description used when the route is bound to an IFC.)
- *rt_if* (*rt_if* can be set to the dotted decimal equivalent of INADDR_ANY, which is 0.)

In addition, the *rt_flags* bit flags can be set to the following:

- RTF_NOREBIND_IFC_FAIL if no rebinding of the route is to occur when the interface associated with the route fails.
- RTF_NOREBIND_IFC_ACTV if no rebinding is to occur when interfaces are activated or deactivated.

To delete a route, the following fields must be set:

- *rt_dst*
- *rt_mask*
- *rt_gateway*
- *rt_tos*
- *rt_protocol*

All other fields are ignored when adding or removing an entry.

SIOCATMARK
Return the value indicating whether socket's read pointer is currently at the out-of-band mark.

The third parameter represents a pointer to an integer flag. If the socket's read pointer is currently at the out-of-band mark, the flag is set to a nonzero value. If it is not, the flag is set to zero.

SIOCDELRT
Delete an entry from the interface routing table. Valid for sockets with address family of `AF_INET`.

See *SIOCADDRT* (page 143) for more information on the third parameter.

SIOCGIFADDR
Get the interface address. Valid for sockets with address family of `AF_INET`.

The third parameter represents a pointer to the structure **ifreq**, defined in **<net/if.h>**:

```
struct ifreq {
   char ifr_name[IFNAMSIZE];
   union {
    struct sockaddr ifru_addr;
    struct sockaddr ifru_mask;
    struct sockaddr ifru_broadaddr;
    short ifru_flags;
    int ifru_mtu;
    int infu_rbufsize;
    char ifru_linename[10];
    char ifru_TOS;
   } ifr_ifru;
};
```

ifr_name is the name of the interface for which information is to be retrieved. The i5/OS implementation requires this field to be set to a NULL-terminated string that represents the interface IP address in dotted decimal format. Depending on the request, one of the fields in the *ifr_ifru* union will be set upon return from the *ioctl()* call. *ifru_addr* is the local IP address of the interface. *ifru_mask* is the subnetwork mask associated with the interface. *ifru_broadaddr* is the broadcast address. *ifru_flags* contains flags that give some information about an interface (for example, token-ring routing support, whether interface is active, broadcast address, and so on). *ifru_mtu* is the maximum transfer unit configured for the interface. *ifru_rbufsize* is the reassembly buffer size of the interface. *ifru_linename* is the line name associated with the interface. *ifru_TOS* is the type of service configured for the interface.

SIOCGIFBRDADDR
Get the interface broadcast address. Valid for sockets with address family of `AF_INET`.

See *SIOCGIFADDR* (page 144) for more information on the third parameter.

SIOCGIFCONF
Get the interface configuration list. Valid for sockets with address family of AF_INET.

The third parameter represents a pointer to the structure **ifconf**, defined in **<net/if.h>**:

```
struct ifconf [
   int ifc_len;
   int ifc_configured;
   int ifc_returned;
   union {
    caddr_t ifcu_buf;
    struct ifreq *ifcu_req;
   } ifc_ifcu;
];
```

ifc_len is a value-result field. The caller passes the size of the buffer pointed to by *ifcu_buf*. On return, *ifc_len* contains the amount of storage that was used in the buffer pointed to by *ifcu_buf* for the interface entries. *ifc_configured* is the number of interface entries in the interface list. *ifc_returned* is the number of interface entries that were returned (this is dependent on the size of the buffer pointed to by *ifcu_buf*). *ifcu_buf* is the user buffer in which a list of interface entries will be stored. Each stored entry will be an *ifreq* structure.

To get the interface configuration list, the following fields must be set:

- *ifc_len*
- *ifcu_buf*

See SIOCGIFADDR (page 144) for more information on the list of *ifreq* structures returned. For this request, the *ifr_name* and *ifru_addr* fields will be set to a value.

Note: Additional information about each individual interface can be obtained using these values and the other interface-related requests.

SIOCGIFFLAGS
Get interface flags. Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 144) for more information on the third parameter.

SIOCGIFLIND
Get the interface line description name. Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 144) for more information on the third parameter.

SIOCGIFMTU
Get the interface network MTU. Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 144) for more information on the third parameter.

SIOCGIFNETMASK
Get the mask for the network portion of the interface address. Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 144) for more information on the third parameter.

SIOCGIFRBUFS
Get the interface reassembly buffer size. Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 144) for more information on the third parameter.

SIOCGIFTOS
Get the interface type-of-service (TOS). Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 144) for more information on the third parameter.

SIOCGPGRP
Get the process ID or process group ID that is to receive the SIGIO and SIGURG signals.

See *FIOGETOWN* (page 142) for more information on the third parameter.

SIOCGRTCONF

Get the route configuration list. Valid for sockets with address family of AF_INET.

For the SIOCGRTCONF request, the third parameter represents a pointer to the structure **rtconf**, also defined in **<net/route.h>**:

```
struct rtconf [
   int rtc_len;
   int rtc_configured;
   int rtc_returned;
   union {
    caddr_t rtcu_buf;
    struct rtentry *rtcu_req;
   } rtc_rtcu;
];
```

rtc_len is a value-result field. The caller passes the size of the buffer pointed to by *rtcu_buf*. On return, *rtc_len* contains the amount of storage that was used in the buffer pointed to by *rtcu_buf* for the route entries. *rtc_configured* is the number of route entries in the route list. *rtc_returned* is the number of route entries that were returned (this is dependent on the size of the buffer pointed to by *rtcu_buf*). *rtcu_buf* is the user buffer in which a list of route entries will be stored. Each stored entry will be an *rtentry* structure.

To get the route configuration list, the following fields must be set:

- *rtc_len*
- *rtcu_buf*

See SIOCADDRT (page 143) for more information on the list of *rtentry* structures returned. For this request, all fields in each *rtentry* structure will be set to a value.

SIOCSENDQ

Return the number of bytes on the send queue that have not been acknowledged by the remote system. Valid for sockets with address family of AF_INET or AF_INET6 and socket type of SOCK_STREAM.

The third parameter represents a pointer to an integer that is set to the number of bytes yet to be acknowledged as being received by the remote TCP transport driver.

Notes:

1. SIOCSENDQ is used after a series of blocking or non-blocking send operations to see if the sent data has reached the transport layer on the remote system. Note that this does not not guarantee the data has reached the remote application.
2. When SIOCSENDQ is used in a multithreaded application, the actions of other threads must be considered by the application. SIOCSENDQ provides a result for a socket descriptor at the given point in time when the *ioctl()*) request is received by the TCP transport layer. Blocking send operations that have not completed, as well as non-blocking send operations in other threads issued after the SIOCSENDQ *ioctl()*, are not reflected in the result obtained for the SIOCSENDQ *ioctl()*.
3. In a situation where the application has multiple threads sending data on the same socket descriptor, the application should not assume that all data has been received by the remote side when 0 is returned if the application is not positive that all send operations in the other threads were complete at the time the SIOCSENDQ *ioctl()* was issued. An application should issue the SIOCSENDQ *ioctl()* only after it has completed all of the send operations. No value is added by querying the machine to see if it has sent all of the data when the application itself has not sent all of the data in a given unit of work.

SIOCSPGRP

Set the process ID or process group ID that is to receive the SIGIO and SIGURG signals.

See *FIOSETOWN* (page 143) for more information on the third parameter.

Authorities

No authorization is required.

Return Value

0	**ioctl()** was successful
-1	**ioctl()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **ioctl()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ENOBUFS (page 542)]	
[ENOSPC (page 541)]	
[ENOSYS (page 544)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[EPERM (page 540)]	
[EPIPE (page 543)]	
[ERESTART (page 547)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNATCH (page 543)]	
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	

Error condition	Additional information
[ETIMEDOUT (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPFA0D4 E	File system error occurred. Error number &1.
CPFA081 E	Unable to set return value or error code.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. QDLS File System Differences

   QDLS does not support **ioctl()**.
3. QOPT File System Differences

   QOPT does not support **ioctl()**.
4. A program must have the appropriate privilege `*IOSYSCFG` to issue any of the following requests: `SIOCADDRT` and `SIOCDELRT`.

Related Information

- The <**sys/ioctl.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- Socket Programming

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

lchown()—Change Owner and Group of Symbolic Link

Syntax

```
#include <unistd.h>

int lchown(const char *path, uid_t owner, gid_t group);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 152.

The **lchown()** function changes the owner and group of a file. If the named file is a symbolic link, **lchown()** changes the owner or group of the link itself rather than the object to which the link points. The permissions of the previous owner or primary group to the object are revoked.

If the file is checked out by another user (someone other than the user profile of the current job), **lchown()** fails with the [EBUSY] error.

When **lchown()** completes successfully, it updates the change time of the file.

Parameters

path (Input) A pointer to the null-terminated path name of the file whose owner and group are being changed.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See “QlgLchown()—Change Owner and Group of Symbolic Link (using NLS-enabled path name)” on page 261 for a description and an example of supplying the *path* in any CCSID.

owner (Input) The user ID (UID) of the new owner of the file. If the value is -1, the user ID is not changed.

group (Input) The group ID (GID) of the new group for the file. If the value is -1, the group ID is not changed.

Note: Changing the owner or the primary group causes the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode to be cleared, unless the caller has *ALLOBJ special authority. If the caller does have *ALLOBJ special authority the bits are not changed. This does not apply to directories. See the **chmod()** documentation.

Authorities

Note: Adopted authority is not used.

Authorization Required for lchown() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object, when changing the owner	Owner and *OBJEXIST (also see **Note 1**)	EPERM
Object, when changing the primary group	See **Note 2**	EPERM
Previous owner’s user profile, when changing the owner	*DLT	EPERM
New owner’s user profile, when changing the owner	*ADD	EPERM

Object Referred to	Authority Required	errno
User profile of previous primary group, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM

Note:

1. You do not need the listed authority if you have *ALLOBJ special authority.
2. At least one of the following must be true:
   a. You have *ALLOBJ special authority.
   b. You are the owner and either of the following:
      - The new primary group is the primary group of the job.
      - The new primary group is one of the supplementary groups of the job.

Authorization Required for lchown() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X See **Note 1**	EACCES
Object when changing the owner	See **Note 2**(a)	EPERM
Object when changing the primary group	See **Note 2**(b)	EPERM

Note:

1. For *FILE objects (such as DDM file, diskette file, print file, and save file), *RX authority is required to the parent directory of the object, rather than just *X authority.
2. The required authorization varies for each object type. For details of the following commands see the iSeries Security Reference book.
   a. CHGOWN
   b. CHGPGP

Authorization Required for lchown() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	*ALLOBJ Special Authority or Owner	EPERM
Previous owner's user profile, when changing the owner	*DLT	EPERM
New owner's user profile, when changing the owner	*ADD	EPERM
Previous primary group's user profile, when changing the primary group	*DLT	EPERM
New primary group's user profile, when changing the primary group	*ADD	EPERM

Authorization Required for lchown() in the QOPT File System

Object Referred to	Authority Required	errno
Volume authorization list	*CHANGE	EACCES
Each directory in the path name preceding the object.	*X	EACCES

Object Referred to	Authority Required	errno
Object	*ALLOBJ Special Authority or Owner	EPERM

Return Value

0 **lchown()** was successful.

-1 **lchown()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **lchown()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs uid or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	For example, *owner* or *group* is not a valid user ID (UID) or group ID (GID).*owner* is the current primary group of the object.
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   lchown() is not supported for member (.MBR) objects.
3. QDLS File System Differences

   The owner and primary group of the /QDLS directory (root folder) cannot be changed. If an attempt is made to change the owner and primary group, a [ENOTSUP] error is returned.
4. QOPT File System Differences

   Changing the owner and primary group is allowed only for an object that exists on a volume formatted in Universal Disk Format (UDF). For all other media formats, ENOTSUP will be returned.

   QOPT file system objects that have owners will not be recognized by the Work with Objects by Owner (WRKOBJOWN) CL command. Likewise, QOPT objects that have a primary group will not be recognized by the Work Objects by Primary Group (WRKOBJPGP) CL command.
5. QFileSvr.400 File System Differences

   The QFileSvr.400 file system does not support **lchown()**.
6. QNetWare File System Differences

   The QNetWare file system does not support primary group. The GID must be zero.
7. QNTC File System Differences

   The owner of files and directories cannot be changed. All files and directories in QNTC are owned by the QDFTOWN user profile.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**limits.h**> file
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations

- “fchown()—Change Owner and Group of File by Descriptor” on page 72—Change Owner and Group of File by Descriptor
- “fstat()—Get File Information by Descriptor” on page 95—Get File Information by Descriptor
- “lstat()—Get File or Link Information” on page 162—Get File or Link Information
- “stat()—Get File Information” on page 468—Get File Information
- “QlgLchown()—Change Owner and Group of Symbolic Link (using NLS-enabled path name)” on page 261—Change Owner and Group of Symbolic Link

Example

See Code disclaimer information for information pertaining to code examples.

The following example changes the owner and group of a file:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

main() {
  char link_name[]="temp.link";
  char fn[]="temp.file";
  struct stat info;

  if (symlink(fn, link_name) == -1)
    perror("symlink() error");
  else {
    lstat(link_name, &info);
    printf("original owner was %d and group was %d\n", info.st_uid,
           info.st_gid);
    if (lchown(link_name, 152, 0) != 0)
      perror("lchown() error");
    else {
      lstat(link_name, &info);
      printf("after lchown(), owner is %d and group is %d\n",
             info.st_uid, info.st_gid);
    }
    unlink(link_name);
  }
}
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

link()—Create Link to File

Syntax

```
#include <unistd.h>

int link(const char *existing, const char *new);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 156.

The **link()** function provides an alternative path name for the existing file, so that the file can be accessed by either the existing name or the new name. **link()** creates a link with a path name *new* to an existing file whose path name is *existing*. The link can be stored in the same directory as the original file or in a different directory.

The **link()** function creates a hard link, which guarantees the existence of a file even after the original path name has been removed.

If **link()** successfully creates the link, it increments the *link count* of the file. The link count indicates how many links there are to the file. If **link()** fails for some reason, the link count is not incremented.

If the *existing* argument names a symbolic link, **link()** creates a link that refers to the file that results from resolving the path name contained in the symbolic link. If *new* names a symbolic link, **link()** fails and sets *errno* to [EEXIST].

A successful link updates the change time of the file, and the change time and modification time of the directory that contains *new* (parent directory).

If the file is checked out by another user (a user profile other than the user profile of the current job), **link()** fails with the [EBUSY] error.

Links created by this function are not allowed to cross file systems. For example, you cannot create a link to a file in the QOpenSys directory from the ″root″ (/) directory.

Links are not allowed to directories. If *existing* names a directory, **link()** fails and sets *errno* to [EPERM].

A job must have access to a file to link to it.

Parameters

existing
: (Input) A pointer to a null-terminated path name naming an existing file to which a new link is to be created.

  This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

  See “QlgLink()—Create Link to File (using NLS-enabled path name)” on page 263 for a description and an example of supplying the *existing* in any CCSID.

new
: (Input) A pointer to a null-terminated path name that is the name of the new link.

  This parameter is assumed to be represented in the CCSID currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job. The new link name is assumed to be represented in the language and country or region currently in effect for the job.

  See “QlgLink()—Create Link to File (using NLS-enabled path name)” on page 263 for a description and an example of supplying the *new* in any CCSID.

Authorities

Note: Adopted authority is not used.

Authorization Required for link()

Object Referred to	Authority Required	errno
Each directory in the *existing* path name that precedes the object being linked to	*X	EACCES
Existing object	*OBJEXIST	EACCES
Each directory in the *new* path name that precedes the object being linked to	*X	EACCES

Object Referred to	Authority Required	errno
Parent directory of the new link	*WX	EACCES

Return Value

0 **link()** was successful.

-1 **link()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **link()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EEXIST (page 543)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EISDIR (page 544)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[EMLINK (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOSYS (page 544)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	Links to directories are not supported.
[EROOBJ (page 545)]	

Error condition	Additional information
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	
[EXDEV (page 541)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. The **link()** function should be used sparingly to avoid potential performance degradation. The greater the number of hard links to an object, the more time it will take to change the attributes of the object.
3. The following file systems do not support **link()**:
   - QSYS.LIB
   - Independent ASP QSYS.LIB
   - QDLS
   - QOPT
   - QFileSvr.400
   - QNetWare
   - QNTC

   If **link()** is used in any of these file systems, a [ENOSYS] error is returned.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "QlgLink()—Create Link to File (using NLS-enabled path name)" on page 263—Create Link to File

- “rename()—Rename File or Directory” on page 460—Rename File or Directory
- “unlink()—Remove Link to File” on page 492—Remove Link to File

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **link()**:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>


main()
{
  char fn[]="link.example.file";
  char ln[]="link.example.link";
  int file_descriptor;
  struct stat info;

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
    puts("before link()");
    stat(fn,&info);
    printf("   number of links is %hu\n",info.st_nlink);
    if (link(fn, ln) != 0) {
      perror("link() error");
      unlink(fn);
    }
    else {
      puts("after link()");
      stat(fn,&info);
      printf("   number of links is %hu\n",info.st_nlink);
      unlink(ln);
      puts("after first unlink()");
      stat(fn,&info);
      printf("   number of links is %hu\n",info.st_nlink);
      unlink(fn);
    }
  }
}
```

Output:

```
before link()
   number of links is 1
after link()
   number of links is 2
after first unlink()
   number of links is 1
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

lseek()—Set File Read/Write Offset

Syntax

```
#include <unistd.h>

off_t lseek(int file_descriptor, off_t offset, int whence);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 160.

The **lseek()** function changes the current file offset to a new position in the file. The new position is the given byte *offset* from the position specified by *whence*. After you have used **lseek()** to seek to a new location, the next I/O operation on the file begins at that location.

lseek() lets you specify new file offsets past the current end of the file. If data is written at such a point, read operations in the gap between this data and the old end of the file will return bytes containing binary zeros (or bytes containing blanks in the QSYS.LIB and independent ASP QSYS.LIB file systems). In other words, the gap is assumed to be filled with zeros (or with blanks in the QSYS.LIB and independent ASP QSYS.LIB file systems). Seeking past the end of a file, however, does not automatically extend the length of the file. There must be a write operation before the file is actually extended.

There are some important considerations for **lseek()** if the O_TEXTDATA and O_CCSID flags were specified on the **open()**, the file CCSID and open CCSID are not the same, and the converted data could expand or contract:

- Making assumptions about data size and the current file offset is extremely dangerous. For example, a file might have a physical size of 100 bytes, but after an application has read 100 bytes from the file, the current file offset may be only 50. To read the whole file, the application might have to read 200 bytes or more, depending on the CCSIDs involved. Therefore, **lseek()** will only be allowed to change the current file offset to:

  – The start of the file (*offset* 0, *whence* SEEK_SET)
  – The end of the file (*offset* 0, *whence* SEEK_END). In this case, the function will return a calculated value based on the physical size of the file, the CCSID of the file, and the CCSID of the open instance. This may be different than the actual file offset.

  If any other combination of values is specified, **lseek()** fails and *errno* is set to ENOTSUP.
- Internally-buffered data from a read or write operation is discarded. See "read()—Read from Descriptor" on page 437 and "write()—Write to Descriptor" on page 502 for more information concerning internal buffering of text data.
- The expected state for the current text conversion is reset to the initial state. This consideration applies only when using a CCSID that can represent data using more than one graphic character set or containing characters of different byte lengths. Some CCSIDs require an escape or shift sequence to signify a state change from one character set or byte length to another. Failing to account for this consideration could lead to incorrect text conversion if, for instance, a double-byte character at the new file offset was treated as two single-byte characters by the conversion function.

In the QSYS.LIB file and independent ASP QSYS.LIB file systems, you can seek only to the beginning of a member while in text mode.

Parameters

file_descriptor
(Input) The file whose current file offset you want to change.

offset
(input) The amount (positive or negative) the byte offset is to be changed. The sign indicates whether the offset is to be moved forward (positive) or backward (negative).

whence
(Input) One of the following symbols (defined in the <**unistd.h**> header file):

SEEK_SET
: The start of the file

SEEK_CUR
: The current file offset in the file

SEEK_END
: The end of the file

If bits in *whence* are set to values other than those defined above, **lseek()** fails with the [EINVAL] error.

Authorities

No authorization is required. Authorization is verified during **open()** or **creat()**.

Return Value

value **lseek()** was successful. The value returned is the new file offset, measured in bytes from the beginning of the file.

-1 **lseek()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **lseek()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ENOENT (page 540)]	
[ENOSPC (page 541)]	
[ENOSYSRSC (page 545)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EOVERFLOW (page 546)]	The resulting file offset would be a value that cannot be represented correctly in a variable of type off_t (the offset is greater than 2GB minus 2 bytes).
[ESPIPE (page 544)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for

another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations (several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data).

3. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   This function is not supported for save files and will fail with error code [ENOTSUP].
4. This function will fail with the [EOVERFLOW] error if the resulting file offset would be a value that cannot be represented correctly in a variable of type off_t (the offset is greater than 2 GB minus 2 bytes).
5. When you develop in C-based languages and an application is compiled with the _LARGE_FILES macro defined, the **lseek()** API will be mapped to a call to the **lseek64()** API. Additionally, the data type off_t will be mapped to the type off64_t.
6. Using this function with the write(), pwrite(), and pwrite64() functions on the /dev/null or /dev/zero character special file will not result in the file data size changing from zero.

Related Information

- The <**unistd.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “creat()—Create or Rewrite File” on page 40—Create or Rewrite File
- “dup()—Duplicate Open File Descriptor” on page 55—Duplicate Open File Descriptor
- “fclear()—Write (Binary Zeros) to Descriptor” on page 77—Write (Binary Zeros) to Descriptor
- “fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)” on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)
- “fcntl()—Perform File Control Command” on page 82—Perform File Control Command
- “lseek64()—Set File Read/Write Offset (Large File Enabled)”—Set File Read/Write Offset (Large File Enabled)
- “open()—Open File” on page 195—Open File
- “pread()—Read from Descriptor with Offset” on page 223—Read from Descriptor with Offset
- “pread64()—Read from Descriptor with Offset (large file enabled)” on page 228—Read from Descriptor with Offset (large file enabled)
- “pwrite()—Write to Descriptor with Offset” on page 229—Write to Descriptor with Offset
- “pwrite64()—Write to Descriptor with Offset (large file enabled)” on page 234—Write to Descriptor with Offset (large file enabled)
- “read()—Read from Descriptor” on page 437—Read from Descriptor
- “write()—Write to Descriptor” on page 502—Write to Descriptor

Example

See Code disclaimer information for information pertaining to code examples.

The following example positions a file (that has at least 11 bytes) to an offset of 10 bytes before the end of the file:

```
lseek(file_descriptor,-10,SEEK_END);
```

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

lseek64()—Set File Read/Write Offset (Large File Enabled)

Syntax

```
#include <unistd.h>

off64_t lseek64(int file_descriptor,
                 off64_t offset, int whence);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes."

The **lseek64()** function changes the current file offset to a new position in the file. The new position is the given byte *offset* from the position specified by *whence*. After you have used **lseek64()** to seek to a new location, the next I/O operation on the file begins at that location.

lseek64() lets you specify new file offsets past the current end of the file. If data is written at such a point, read operations in the gap between this data and the old end of the file will return bytes containing binary zeros (or bytes containing blanks in the QSYS.LIB or independent ASP QSYS.LIB file systems). In other words, the gap is assumed to be filled with zeros (or with blanks in the QSYS.LIB or independent ASP QSYS.LIB file systems). If you seek past the end of a file, however, the length of the file is not automatically extended. The maximum offset that can be specified is the largest value that can be held in an 8-byte, signed integer. You must do a write operation before the file is actually extended.

In the QSYS.LIB or independent ASP QSYS.LIB file systems, you can seek only to the beginning of a member while in text mode.

lseek64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:

- Using the **open64()** function (see "open64()—Open File (Large File Enabled)" on page 211).
- Using the **open()** function (see "open()—Open File" on page 195) with the O_LARGEFILE flag set.

For additional information about parameters, authorities required, error conditions and examples, see "lseek()—Set File Read/Write Offset" on page 157.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **lseek64()** API and the off64_t data type, you must compile the source with the _LARGE_FILE_API defined.
2. All of the usage notes for **lseek()** apply to **lseek64()**. See "Usage Notes" on page 160 in the **lseek()** API.

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

---

lstat()—Get File or Link Information

Syntax

```
#include <sys/stat.h>

int lstat(const char *path, struct stat *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 165.

The **lstat()** function gets status information about a specified file and places it in the area of memory pointed to by *buf*. If the named file is a symbolic link, **lstat()** returns information about the symbolic link itself.

The information is returned in the `stat` structure, referenced by *buf*. For details on the `stat` structure, see "stat()—Get File Information" on page 468.

If the named file is not a symbolic link, **lstat()** updates the time-related fields before putting information in the `stat` structure.

Parameters

path (Input) A pointer to the null-terminated path name of the file.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgLstat()—Get File or Link Information (using NLS-enabled path name)" on page 265 for a description and an example of supplying the *path* in any CCSID.

buf (Output) A pointer to the area to which the information should be written.

Authorities

Note: Adopted authority is not used.

Authorization Required for lstat()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object, if object type is not *USRPRF	None	None
Object, if object type is *USRPRF	Any authority greater than *EXCLUDE	ENOENT

Return Value

0 **lstat()** was successful. The information is returned in *buf*.

-1 **lstat()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **lstat()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	

Error condition	Additional information
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOSPC (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EOVERFLOW (page 546)]	The file size in bytes cannot be represented correctly in the structure pointed to by *buf* (the file is larger than 2GB minus 1 byte).
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.

Message ID	Error Message Text
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:

   - Where multiple threads exist in the job.
   - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. QOPT File System Differences

   The value for st_atime will always be zero. The value for st_ctime will always be the creation date and time of the file or directory.

   If the object exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the object and each directory in the path name follows the rules described in Authorization Required for lstat() (page 163). If the object exists on a volume formatted in some other media format, no authorization checks are made on the object or each directory in the path name. The volume authorization list is checked for *USE authority regardless of the volume media format.

   The user, group, and other mode bits are always on for an object tha exists on a volume not formatted in Universal Disk format (UDF).

   lstat() on /QOPT will always return 2,147,483,647 for size fields.

   lstat() on optical volumes will return the volume capacity or 2,147,483,647, whichever is smaller.

   The file access time is not changed.
3. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)
4. QNetWare File System Differences

   The QNetWare file system does not fully support mode bits. See NetWare on iSeries for more information.

5. This function will fail with the [EOVERFLOW] error if the file size in bytes cannot be represented correctly in the structure pointed to by buf (the file is larger than 2GB minus 1 byte).
6. When you develop in C-based languages and this function is compiled with _LARGE_FILES defined, it will be mapped to **lstat64()**. Note that the type of the buf parameter, `struct stat`, also will be mapped to type `struct stat64`.

Related Information

- The <**sys/stat.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "chown()—Change Owner and Group of File" on page 29—Change Owner and Group of File
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "dup()—Duplicate Open File Descriptor" on page 55—Duplicate Open File Descriptor
- "fclear()—Write (Binary Zeros) to Descriptor" on page 77—Write (Binary Zeros) to Descriptor
- "fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)" on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "fstat()—Get File Information by Descriptor" on page 95—Get File Information by Descriptor
- "link()—Create Link to File" on page 153—Create Link to File
- "mkdir()—Make Directory" on page 169—Make Directory
- "open()—Open File" on page 195—Open File
- "QlgLstat()—Get File or Link Information (using NLS-enabled path name)" on page 265—Get File or Link Information
- "read()—Read from Descriptor" on page 437—Read from Descriptor
- "readlink()—Read Value of Symbolic Link" on page 452—Read Value of Symbolic Link
- "stat()—Get File Information" on page 468—Get File Information
- "symlink()—Make Symbolic Link" on page 485—Make Symbolic Link
- "unlink()—Remove Link to File" on page 492—Remove Link to File
- "utime()—Set File Access and Modification Times" on page 497—Set File Access and Modification Times
- "write()—Write to Descriptor" on page 502—Write to Descriptor

Example

See Code disclaimer information for information pertaining to code examples.

The following example provides status information for a file:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>

main() {
  char fn[]="temp.file", ln[]="temp.link";
  struct stat info;
  int file_descriptor;

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
```

```
    close(file_descriptor);
    if (link(fn, ln) != 0)
      perror("link() error");
    else {
      if (lstat(ln, &info) != 0)
        perror("lstat() error");
      else {
        puts("lstat() returned:");
        printf("  inode:   %d\n",   (int) info.st_ino);
        printf(" dev id:   %d\n",   (int) info.st_dev);
        printf("   mode:   %08x\n",       info.st_mode);
        printf("  links:   %d\n",         info.st_nlink);
        printf("    uid:   %d\n",   (int) info.st_uid);
        printf("    gid:   %d\n",   (int) info.st_gid);
      }
      unlink(ln);
    }
    unlink(fn);
  }
}
```

Output:

```
lstat() returned:
  inode:   3022
 dev id:   1
   mode:   00008080
  links:   2
    uid:   137
    gid:   500
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

lstat64()—Get File or Link Information (Large File Enabled)

Syntax

```
#include <sys/stat.h>

int lstat64(const char *path, struct stat64 *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "lstat()—Get File or Link Information" on page 162.

The **lstat64()** function gets status information about a specified file and places it in the area of memory pointed to by *buf*. If the named file is a symbolic link, **lstat64()** returns information about the symbolic link itself.

The information is returned in the `stat64` structure, referred to by *buf*. For details on the `stat64` structure, see "stat64()—Get File Information (Large File Enabled)" on page 475.

If the named file is not a symbolic link, **lstat64()** updates the time-related fields before putting information in the `stat64` structure.

For additional information about parameters, authorities required, and error conditions, see "lstat()—Get File or Link Information" on page 162.

See "QlgLstat64()—Get File or Link Information (large file enabled and using NLS-enabled path name)" on page 267 for a description and an example of supplying the *path* in any CCSID.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **lstat64()** API and the `struct stat64` data type, you must compile the source with the _LARGE_FILE_API defined.
2. All of the usage notes for **lstat()** apply to **lstat64()**. See "Usage Notes" on page 165 in the **lstat()** API.

Example

See Code disclaimer information for information pertaining to code examples.

The following example provides status information for a file.

```
#define _LARGE_FILE_API
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>

main() {
  char fn[]="temp.file", ln[]="temp.link";
  struct stat64 info;
  int file_descriptor;

  if ((file_descriptor = creat64(fn, S_IWUSR)) < 0)
    perror("creat64() error");
  else {
    close(file_descriptor);
    if (link(fn, ln) != 0)
      perror("link() error");
    else {
      if (lstat64(ln, &info) != 0)
        perror("lstat64() error");
      else {
        puts("lstat64() returned:");
        printf("  inode:   %d\n",   (int) info.st_ino);
        printf(" dev id:   %d\n",   (int) info.st_dev);
        printf("   mode:   %08x\n",       info.st_mode);
        printf("  links:   %d\n",         info.st_nlink);
        printf("    uid:   %d\n",   (int) info.st_uid);
        printf("    gid:   %d\n",   (int) info.st_gid);
        printf("   size:   %lld\n", (long long) info.st_size);
      }
      unlink(ln);
    }
    unlink(fn);
  }
}
```

Output:

```
lstat() returned:
  inode:   3022
 dev id:   1
   mode:   00008080
  links:   2
    uid:   137
    gid:   500
   size:   18
```

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

mkdir()—Make Directory

Syntax

```
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 172.

The **mkdir()** function creates a new, empty directory whose name is defined by *path*. The file permission bits in *mode* are modified by the file creation mask of the job and then used to set the file permission bits of the directory being created.

For more information on the permission bits in *mode* see "chmod()—Change File Authorizations" on page 22. For more information on the file creation mask, see "umask()—Set Authorization Mask for Job" on page 491.

The owner ID of the new directory is set to the effective user ID (uid) of the job. If the directory is being created in the "root" (/), QOpenSys, and user-defined file systems, the following applies. If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the directory. If the S_ISGID bit of the parent directory is on, the group ID (GID) of the new directory is set to the GID of the parent directory. For all other file systems, the group ID (GID) of the new directory is set to the GID of the parent directory.

mkdir() sets the access, change, modification, and creation times for the new directory. It also sets the change and modification times for the directory that contains the new directory (parent directory).

The link count of the parent directory link count is increased by one. The link count of the new directory is set to 2. The new directory also contains an entry for "dot" (.) and "dot-dot" (..).

If *path* names a symbolic link, the symbolic link is not followed, and **mkdir()** fails with the [EEXIST] error.

If bits in *mode* other than the file permission bits are set, **mkdir()** fails with the [EINVAL] error.

Parameters

path
(Input) A pointer to the null-terminated path name of the directory to be created.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

The name of the new directory is assumed to be represented in the language and country or region currently in effect for the process.

See "QlgMkdir()—Make Directory (using NLS-enabled path name)" on page 270 for a description and an example of supplying the *path* in any CCSID.

mode
(Input) Permission bits for the new directory. The S_ISVTX bit may also be specified when creating the directory.

See "chmod()—Change File Authorizations" on page 22 for details on the values that can be specified for *mode*.

Authorities

Note: Adopted authority is not used.

Authorization Required for mkdir() (excluding QSYS.LIB, Independent ASP QSYS.LIB, and QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be created.	*X	EACCES
Parent directory of directory to be created	*WX	EACCES

Authorization Required for mkdir() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be created.	*X	EACCES
Parent directory of directory to be created (when the directory being created is a database file)	*X and *ADD	EACCES

Authorization Required for mkdir() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be created.	*X	EACCES
Parent directory of directory to be created	*CHANGE	EACCES

Return Value

0 **mkdir()** was successful. The directory was created.

-1 **mkdir()** was not successful. The directory was not created. The *errno* global variable is set to indicate the error.

Error Conditions

If **mkdir()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EEXIST (page 543)]	The named file, directory, or path already exists. Or, the last component of *path* is a symbolic link.
[EFAULT (page 541)]	

Error condition	Additional information
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[EMLINK (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOSYS (page 544)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - There are secondary threads active in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. "Root" (/), QOpenSys, and User-Defined File System Differences

   The user who creates the directory becomes its owner.

   The S_ISGID bit of the directory affects what the group ID (GID) is for objects that are created in the directory. If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the S_ISGID bit of the parent directory is on, the group ID (GID) is copied from the parent directory in which the new directory is being created.

   The owner, primary group, and public object authorities (*OBJEXIST, *OBJMGT, *OBJALTER, and *OBJREF) are copied from the parent directory's owner, primary group, and public object authorities. This occurs even when the new directory has a different owner than the parent directory. The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except for those permissions that are also set in the file mode creation mask). The new directory does not have any private authorities or authorization list. It only has authorities for the owner, primary group, and public.

   The create object scanning attribute value for this directory is copied from the create object scanning attribute value of the parent directory. For more information on this attribute, see "Qp0lSetAttr()—Set Attributes" on page 403—Set Attributes.

   The create object auditing attribute value for this directory will be set to *SYSVAL. For more information on this attribute, see "Qp0lSetAttr()—Set Attributes" on page 403—Set Attributes.

3. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   The user who creates the directory becomes its owner. The group ID is copied from the primary user ID, if one exists.

   The owner is given *ALL object authority to the new directory. The group object authorities are copied from the user profile of the owner. The public receives no object authority to the directory.

   The primary group authorities specified in *mode* are not saved if no primary group exists.

The change and modification times for the directory that contains the new directory are only set when the new directory is a database file.

» The create object auditing attribute value for this directory will be set to *SYSVAL. For more information on this attribute, see “Qp0lSetAttr()—Set Attributes” on page 403—Set Attributes.

4. QDLS File System Differences

   The user who creates the directory becomes its owner. The group ID is copied from the parent folder in which the new directory is being created.

   The object authority of the owner is set to *OBJMGT + *OBJEXIST + *OBJALTER + *OBJREF.

   The primary group and public object authority and all other authorities are copied from the parent folder.

   The owner, primary group, and public data authority (including *OBJOPR) are derived from the permissions specified in *mode* (except those permissions that are also set in the file mode creation mask).

   The primary group authorities specified in *mode* are not saved if no primary group exists.

5. QOPT File System Differences

   When the volume on which the directory is being created is formatted in Universal Disk Format (UDF):

   - The authorization that is checked for the object and preceding directories in the path name follows the rules described in Authorization Required for mkdir() (page 170).
   - The volume authorization list is checked for *CHANGE authority.
   - The user who creates the file becomes its owner.
   - The group ID is copied from the parent directory in which the file is created.
   - The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode.
   - The same uppercase and lowercase forms in which the names are entered are preserved. No distinction is made between uppercase and lowercase when searching for names.

   When the volume on which the directory is being created is not formatted in Universal Disk Format (UDF):

   - No authorization is checked on the object or preceding directories in the path name.
   - The volume authorization list is checked for *CHANGE authority.
   - QDFTOWN becomes the owner of the directory.
   - No group ID is assigned to the directory.
   - The permissions specified in the mode are ignored. The owner, primary group, and public data authorities are set to RWX.
   - For newly created directories, names are created in uppercase. No distinction is made between uppercase and lowercase when searching for names.

   A directory cannot be created as a direct child of /QOPT.

   The change and modification times of the parent directory are not updated.

6. Network File System Differences

   Local access to remote directories through the Network File System may produce unexpected results due to conditions at the server. The creation of a directory may fail if permissions and other attributes that are stored locally by the Network File System are more restrictive than those at the server. A later attempt to create a file can succeed when the locally stored data has been refreshed. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) The creation can also succeed after the file system has been remounted.

If you try to re-create a directory that was recently deleted, the request may fail because data that was stored locally by the Network File System still has a record of the directory's existence. The creation succeeds when the locally stored data has been updated.

7. QNetWare File System Differences

   The QNetWare file system does not fully support mode bits. See NetWare on iSeries for more information.

8. QNTC File System Differences

   Directory authorities are inherited from the access control list (if any exists) of the parent directory. The mode bits are ignored.

   In addition to the normal **mkdir()** function, in the QNTC file system, **mkdir()** can be used to add a server directory under the /QNTC directory level. Directories for all functional Windows NT servers in the local subnet are automatically created. However, Windows NT servers outside the local subnet must be added by using **mkdir()** or the MKDIR command. For example:

   ```
   char new_dir[]="/QNTC/NTSRV1";
   mkdir(new_dir,NULL)
   ```

   would add the NTSRV1 server into the QNTC directory structure for future access of files and directories on that server.

   It is also possible to add the server by using the TCP/IP address. For example:

   ```
   char new_dir[]="/QNTC/9.130.67.24";
   mkdir(new_dir,NULL)
   ```

   The directories added using **mkdir()** in the QNTC file system will not persist across IPLs. Thus, **mkdir()** or the Make Directory (MKDIR) command must be reissued after every system IPL.

Related Information

- The <**sys/stat.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "QlgMkdir()—Make Directory (using NLS-enabled path name)" on page 270—Make Directory
- "stat()—Get File Information" on page 468—Get File Information
- "umask()—Set Authorization Mask for Job" on page 491—Set Authorization Mask for Job
- "pathconf()—Get Configurable Path Name Variables" on page 216—Get Configurable Path Name Variables

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a new directory:

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

main() {
  char new_dir[]="new_dir";

  if (mkdir(new_dir, S_IRWXU|S_IRGRP|S_IXGRP) != 0)
    perror("mkdir() error");
  else if (chdir(new_dir) != 0)
    perror("first chdir() error");
  else if (chdir("..") != 0)
    perror("second chdir() error");
  else if (rmdir(new_dir) != 0)
    perror("rmdir() error");
  else
    puts("success!");
}
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

mkfifo()—Make FIFO Special File

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 177.

The **mkfifo()** function creates a new FIFO special file (FIFO) whose name is defined by *path*. A FIFO special file is a type of file with the property that data written to the file is read on a first-in-first-out basis. See the **open()**, **read()**, **write()**, **lseek**, and **close** functions for more characteristics of a FIFO special file.

A FIFO may be opened for reading only or writing only for a uni-directional I/O. It also may be opened for reading and writing access to provide a bi-directional FIFO descriptor.

The file permission bits in *mode* are modified by the file creation mask of the job and then used to set the file permission bits of the FIFO being created.

For more information on the permission bits in *mode*, see "chmod()—Change File Authorizations" on page 22—Change File Authorizations. For more information on the file creation mask, see "umask()—Set Authorization Mask for Job" on page 491—Set Authorization Mask for Job.

The owner ID of the new FIFO is set to the effective user ID (UID) of the thread. If the object is being created in the "root" (/), QOpenSys, and user-defined file systems, the following applies. If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the S_ISGID bit of the parent directory is on, the group ID (GID) of the new object is set to the GID of the parent directory. For all other file systems, the group ID (GID) of the new FIFO is set to the GID of the parent directory.

Upon successful completion, **mkfifo()** sets the access, change, modification, and creation times for the new FIFO. It also sets the change and modification times for the directory that contains the new FIFO (parent directory).

If *path* contains a symbolic link, the symbolic link is followed.

If *path* names a symbolic link, the symbolic link is not followed, and **mkfifo()** fails with the [EEXIST] error.

If bits in *mode* other than the file permission bits are set, **mkfifo()** fails with the [EINVAL] error.

Parameters

path (Input) A pointer to the null-terminated path name of the FIFO special file to be created.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

The name of the new FIFO is assumed to be represented in the language and country or region currently in effect for the process.

See "QlgMkfifo()—Make FIFO Special File (using NLS-enabled path name)" on page 271—Make FIFO Special File (using NLS-enabled path name) for a description and an example of supplying the *path* in any CCSID.

mode (Input) Permission bits for the new FIFO.

Authorities

Adopted authority is not used.

Authorization Required for mkfifo()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the FIFO to be created.	*X	EACCES
Parent directory of FIFO to be created	*WX	EACCES

Return Value

0 **mkfifo()** was successful. The FIFO was created.

-1 **mkfifo()** was not successful. The FIFO was not created. The *errno* global variable is set to indicate the error.

Error Conditions

If **mkfifo()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file also may fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EEXIST (page 543)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ELOOP (page 544)]	
[EMLINK (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	

Error condition	Additional information
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOSYS (page 544)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	
[EROFS (page 544)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.

- The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

  – ″Root″ (/)
  – QOpenSys
  – User-defined
  – QNTC
  – QSYS.LIB
  – Independent ASP QSYS.LIB
  – QOPT
  – QFileSvr.400

2. The following file systems support **mkfifo()**:
   - ″Root″ (/)
   - QOpenSys
   - User-defined
3. There are some restrictions when opening a FIFO for text conversion and the CCSIDs involved are not strictly single-byte:
   - Opening a FIFO for reading or reading and writing is not allowed.
   - Any conversion between CCSIDs that are not strictly single-byte must be done by an open instance that has write-only access.
4. The owner, primary group, and public object authorities (*OBJEXIST, *OBJMGT, *OBJALTER, and *OBJREF) are copied from the parent directory's owner, primary group, and public object authorities. This occurs even when the new FIFO has a different owner than the parent directory. The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except for those permissions that are also set in the file mode creation mask). The new FIFO does not have any private authorities or authorization list. It only has authorities for the owner, primary group, and public.

Related Information

- The <**sys/stat.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "umask()—Set Authorization Mask for Job" on page 491—Set Authorization Mask for Job
- "QlgMkfifo()—Make FIFO Special File (using NLS-enabled path name)" on page 271—Make FIFO Special File (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a new FIFO:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

void main() {
  char *mypath = "/newFIFO";

  if (mkfifo(mypath, S_IRWXU|S_IRWXO) != 0)
    perror("mkfifo() error");
  else
```

```
    puts("success!");

  return;
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

mmap()—Memory Map a File

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

void *mmap( void *addr,
            size_t len,
            int protection,
            int flags,
            int fildes,
            off_t off);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Yes

The mmap() function establishes a mapping between a process' address space and a stream file.

The address space of the process from the address returned to the caller, for a length of *len*, is mapped onto a stream file starting at offset *off*.

The portion of the stream file being mapped is from starting offset *off* for a length of *len* bytes. The actual address returned from the function is derived from the values of *flags* and the value specified for *address*.

The **mmap()** function causes a reference to be associated with the file represented by *fildes*. This reference is not removed by subsequent close operations. The file remains referenced as long as a mapping exists over the file.

If a mapping already exists for the portion of the processes address space that is to be mapped and the value MAP_FIXED was specified for *flags*, then the previous mappings for the affected pages are implicitly unmapped. If one or more files affected by the implicit unmap no longer have active mappings, these files will be unreferenced as a result of **mmap()**.

The use of the **mmap()** function is restricted by the ***QSHRMEMCTL*** System Value. When this system value is 0, the **mmap()** function may not create a shared mapping having with ***PROT_WRITE*** capability. Essentially, this prevents the creation of a memory map that could alter the contents of the stream file being mapped. If the *flags* parameter indicated ***MAP_SHARED***, the ***prot*** parameter specifies ***PROT_WRITE*** and the ***QSHRMEMCTL*** system value is 0, then the **mmap()** functions will fail and an error number of ***EACCES*** results.

When the **mmap()** function creates a memory map, the current value of the ***QSHRMEMCTL*** system value is stored with the mapping. This further restricts attempts to change the protection of the mapping through the use of the **mprotect** function. Changing the system valueonly affects memory maps created after the system value is changed.

If the size of the file increases after the **mmap()** function completes, then the whole pages beyond the original end of file will not be accessible via the mapping.

If the size of the mapped file is decreased after **mmap()**, attempts to reference beyond the end of the file are undefined and may result in an MCH0601 exception.

Any data written to that portion of the file that is allocated beyond end-of-file may not be preserved. Changes made beyond end of file via mapped access may not be preserved.

The portion of the file beyond end-of-file is assumed to be zero by the traditional file access APIs such as **read()**, **readv()**, **write()**, **writev()**, and **ftruncate()**. The system may clear a partial page, or whole pages allocated beyond end-of-file. This must be taken into account when directly changing a memory mapped file beyond end-of-file. It is not recommended that data be directly changed beyond end-of-file because the extra space allocated varies and unpredictable results may occur.

The **mmap()** function is only supported for *TYPE2 stream files (*STMF) existing in the ″root″ (/), QOpenSys, and user-defined file systems.

Journaling cannot be started while a file is memory mapped. Likewise, a journaled file cannot be memory mapped. The **mmap()** function will fail with » ENOTSUP « if the file is journaled.

The *off* parameter must be zero or a multiple of the system page size. The _SC_PAGESIZE or _SC_PAGE_SIZE options on the “sysconf()—Get System Configuration Variables” on page 488 function may be used to retrieve the system page size.

Parameters

addr (Input) The starting address of the memory area to be mapped. If the MAP_FIXED value is specified with the *flag* parameter, then *address* must be a multiple of the system page size. Use the _SC_PAGESIZE or _SC_PAGE_SIZE options of the **sysconf()** API to obtain the actual page size in an implementation-independent manner. When the MAP_FIXED flag is specified, this address must not be zero.

len (Input) The length in bytes to map. A length of zero will result in an errno of EINVAL.

protection

(Input) The access allowed to this process for this mapping. Specify PROT_NONE, PROT_READ, PROT_WRITE, or a the inclusive-or of PROT_READ and PROT_WRITE. You cannot specify a protection value more permissive than the mode in which the file was opened.

The PROT_WRITE value requires that the file be opened for write and read access.

The following table shows the symbolic constants allowed for the protection parameter.

Symbolic Constant	Decimal Value	Description
PROT_READ	1	Read access is allowed.
PROT_WRITE	2	Write access is allowed. Note that this value assumes PROT_READ also.
PROT_NONE	8	No data access is allowed.
PROT_EXEC	4	This value is allowed, but is equivalent to PROT_READ.

flags (Input) Further defines the type of mapping desired. There are actually two independent options controlled through the *flags* parameter.

The first attribute controls whether or not changes made through the mapping will be seen by other processes. The MAP_PRIVATE option will cause a copy on write mapping to be created. A change to the mapping results in a change to a private copy of the affected portion of the file. These changes cannot be seen by other processes. The MAP_SHARED option provides a memory mapping of the file where changes (if allowed by the *protection* parameter) are made to the file. Changes are shared with other processes when MAP_SHARED is specified.

The second control provided by the *flags* parameter in conjunction with the value of the *addr* parameter influences the address range assigned to the mapping. You may request one of the following address selection modes:

1. An exact address range specification. The system will set up the mapping at this location if the address range is valid. Any mapping in the successfully mapping range that existed prior to the mapping operation is implicitly unmapped by this operation.
2. A suggested address range. The system will select a range close to the suggested range.
3. System selected. The system will select an address range. This usually is used to acquire the initial memory map range. Subsequent ranges can be mapped relative to this range.

The MAP_FIXED flag value specifies that the virtual address has been specified through the *addr* parameter. The **mmap()** function will use the value of *addr* as the starting point of the memory map.

When MAP_FIXED is set in the flags parameter, the system is informed that the return value must be equal to the value of *addr*. An invalid value of *addr* when MAP_FIXED is specified will result in a value of MAP_FAILED, which has a value of 0, for the returned value and the the value of errno will be set to EINVAL.

When MAP_FIXED is not specified, a value of zero for parameter *addr* indicates that the system may choose the value for the return value. If MAP_FIXED is not specified and a nonzero value is specified for *addr*, the system will take this as a suggestion to find a contiguous address range close to the specified address.

The following table shows the symbolic constants allowed for the flags parameter.

Symbolic Constant	**Decimal Value**	**Description**
MAP_SHARED	4	Changes are shared.
MAP_PRIVATE	2	Changes are private.
MAP_FIXED	1	Parameter *addr* has exact address

fildes (Input) An open file descriptor.

off (Input) The offset into the file, in bytes, where the map should begin.

Authorities

No authority checking is performed by the **mmap()** function because this was done by the **open()** functions which assigned the file descriptor, *fildes*, used by the **mmap()** function.

The following table shows the open access intent that is required for the various combinations of the mapping sharing mode and mapping intent.

Mapping Sharing Mode	**Mapping Intent**	**Open access intents allowed**
MAP_SHARED	PROT_READ	O_RDONLY or O_RDWR
MAP_SHARED	PROT_WRITE	O_RDWR
MAP_SHARED	PROT_NONE	O_RDONLY or O_RDWR
MAP_PRIVATE	PROT_READ	O_RDONLY or O_RDWR
MAP_PRIVATE	PROT_WRITE	O_RDONLY or O_RDWR
MAP_PRIVATE	PROT_NONE	O_RDONLY or O_RDWR

Return Value

Upon successful completion, the **mmap()** function returns the address at which the mapping was placed; otherwise, it returns a value of MAP_FAILED, which has a value of 0, and sets errno to indicate the error. The symbol MAP_FAILED is defined in the header <sys/mman.h>.

If successful, function **mmap()** will never return a value of MAP_FAILED.

If **mmap()** fails for reasons other than EBADF, EINVAL, or ENOTSUP, some of the mappings in the address range starting at *addr* and continuing for *len* bytes may have been unmapped and no new mappings are created.

Error Conditions

When the **mmap()** function fails, it returns MAP_FAILED, which has a value of 0, and sets the errno as follows.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems. The file referenced by fildes is not open for read, or the file is not opened for write and PROT_WRITE for a shared mapping is being requested. This error also results when the ***QSHRMEMCTL*** system value is 0 and PROT_WRITE is specified.
[EBADFUNC (page 540)]	A given file descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open file.
[EINVAL (page 540)]	The value of the *addr* parameter is not valid. This can occur when MAP_FIXED is specified and the value of the *addr* parameter is not a multiple of the system page size. This may also occur if the value for parameter *addr* is not a valid VOID* pointer or is not within the range allowed. This error number is also returned if the value of the *flags* parameter does not indicate either MAP_SHARED or MAP_PRIVATE.
[ENODEV (page 540)]	The *fildes* parameter does not refer to a *TYPE2 stream file (*STMF) in the "root" (/), QOpenSys, or user-defined file systems.
[ENOMEM (page 543)]	This can occur if the portion of the local process address space reserved for memory mapping has been exceeded. When MAP_FIXED is specified, it may also occur if the address range specified by the combination of the *addr* and *len* parameters results in a range outside the range reserved for process local storage.
[ENOTAVAIL (page 547)]	
[ENOTSUP (page 542)]	
[ENXIO (page 541)]	The portion of the file, as specified by *off* and *len* is not valid for the current size of the file.
[EOVERFLOW (page 546)]	
[EUNKNOWN (page 544)]	

Error Messages

The following messages may be sent from this function.

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred.

Message ID	Error Message Text
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [EBADF] when *fildes* is a scan descriptor that was passed to one of the scan-related exit programs. See "Integrated File System Scan on Open Exit Program" on page 523 and "Integrated File System Scan on Close Exit Program" on page 513 for more information.
2. The **msync()** function must be used to write changed pages of a shared mapping to disk. If a system crash occurs before the **msync** function is executed, some data may not be preserved.
3. If the application chooses to mix file access methods such as **read()**, **readv()**, **write()**, **writev()**, **ftruncate()**, **fclear()** or their related APIs with **mmap()**, then the application must ensure proper synchronization. While operations such as **read()** , **write()** **ftruncate()**, and **fclear()** are relatively atomic because of internal locking, access through the memory map created by **mmap()** does not synchronize with the **read()**, **readv()**, **write()**, **writev()** **ftruncate()**, and **fclear()** functions. Several synchronization functions are available, including the **fcntl()** API, the **DosDetFileLocks()** API, and the mutex functions. Use one of these synchronization methods around access and modifications if atomic access is required. These techniques also will ensure atomic access in a multiprocessor environment.
4. When using **mmap()**, it is necessary to first make a nonspecific mapping request to generate a valid address. This is easily done by specifying a requested address (*addr*) of ***0*** and not specifying *MAP_FIXED*. Then, using the returned address *pa* as the new requested address (*addr*) and also specifying *MAP_FIXED* for the *flags* parameter. The example below illustrates how this technique can be applied to achieve a contiguous mapping of several files.
5. The address pointer returned by **mmap()** can only be used with the V4R4M0 or later versions of the following languages:
   - ILE COBOL
   - ILE RPG
   - ILE C if the TERASPACE parameter is used when compiling the program.
6. The application cannot write or store any data via the memory mapping which includes any tagged (16-byte) pointers because the pointer attribute will be lost. Some examples of tagged pointers include space pointers, system pointers, invocation pointers etc..

   If the DTAMDL(*LLP64) parameter is used when compiling an ILE C program, this limitation does not apply as the pointers will be 8 byte pointers, and their pointer attribute will be preserved.

Related Information

- "DosSetFileLocks()—Lock and Unlock a Byte Range of an Open File" on page 47—Lock and Unlock a Byte Range of an Open File
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "fclear()—Write (Binary Zeros) to Descriptor" on page 77—Write (Binary Zeros) to Descriptor
- "ftruncate()—Truncate File" on page 109—Truncate File
- "mmap64()—Memory map a Stream File (Large File Enabled)" on page 186—Memory Map a Stream File (Large File Enabled)
- "munmap()—Remove Memory Mapping" on page 193—Remove Memory Mapping
- "mprotect()—Change Access Protection for Memory Mapping" on page 186—Change Access Protection for Memory Mapping
- "msync()—Synchronize Modified Data with Mapped File" on page 190—Synchronize Modified Data with Mapped File
- "open()—Open File" on page 195—Open File

- “open64()—Open File (Large File Enabled)” on page 211—Open File (Large File Enabled)
- “read()—Read from Descriptor” on page 437—Read from Descriptor
- “readv()—Read from Descriptor Using Multiple Buffers” on page 455—Read from Descriptor Using Multiple Buffers
- “sysconf()—Get System Configuration Variables” on page 488—Get system configuration variables
- “write()—Write to Descriptor” on page 502—Write to Descriptor
- “writev()—Write to Descriptor Using Multiple Buffers” on page 509—Write to Descriptor Using Multiple Buffers

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates two files and then produces a contiguous memory mapping of the first data page of each file using two invocations of **mmap()**.

```
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/mman.h>

main(void) {

  size_t bytesWritten = 0;
  int   my_offset = 0;
  char  text1Ý="Data for file 1.";
  char  text2Ý="Data for file 2.";
  int fd1,fd2;
  int PageSize;
  void *address;
  void *address2;
  fd1 = open("/tmp/mmaptest1",
            (O_CREAT | O_TRUNC | O_RDWR),
            (S_IRWXU | S_IRWXG | S_IRWXO) );
  if ( fd1 < 0 )
    perror("open() error");
  else {
    bytesWritten = write(fd1, text1, strlen(text1));
    if ( bytesWritten != strlen(text1) ) {
      perror("write() error");
      int closeRC = close(fd1);
      return -1;
    }

  fd2 = open("/tmp/mmaptest2",
            (O_CREAT | O_TRUNC | O_RDWR),
            (S_IRWXU | S_IRWXG | S_IRWXO) );
    if (fd2 < 0 )
      perror("open() error");
    else {
      bytesWritten = write(fd2, text2, strlen(text2));
      if ( bytesWritten != strlen(text2) )
        perror("write() error");

      PageSize = (int)sysconf(_SC_PAGESIZE);
      if ( PageSize < 0) {
        perror("sysconf() error");
      }
      else {
```

```
    off_t lastoffset = lseek( fd1, PageSize-1, SEEK_SET);
    if (lastoffset < 0 ) {
      perror("lseek() error");
    }
    else {
    bytesWritten = write(fd1, " ", 1);   /* grow file 1 to 1 page. */

    off_t lastoffset = lseek( fd2, PageSize-1, SEEK_SET);

    bytesWritten = write(fd2, " ", 1);   /* grow file 2 to 1 page. */
      /*
       *  We want to show how to memory map two files with
       *  the same memory map.  We are going to create a two page
       *  memory map over file number 1, even though there is only
       *  one page available. Then we will come back and remap
       *  the 2nd page of the address range returned from step 1
       *  over the first 4096 bytes of file 2.
       */

     int len;

     my_offset = 0;
     len = PageSize;   /* Map one page */
     address = mmap(NULL,
                    len,
                    PROT_READ,
                    MAP_SHARED,
                    fd1,
                    my_offset );
     if ( address != MAP_FAILED ) {
       address2 = mmap( ((char*)address)+PageSize,
                     len,
                     PROT_READ,
                     MAP_SHARED | MAP_FIXED, fd2,
                     my_offset );
       if ( address2 != MAP_FAILED ) {
             /* print data from file 1 */
          printf("\n%s",address);
             /* print data from file 2 */
          printf("\n%s",address2);
       } /* address 2 was okay. */
       else {
         perror("mmap() error=");
       } /* mmap for file 2 failed. */
     }
     else {
       perror("munmap() error=");
     }
       /*
        *  Unmap two pages.
        */
     if ( munmap(address, 2*PageSize) < 0) {
      perror("munmap() error");
    }
    else;

   }
  }
  close(fd2);
  unlink( "/tmp/mmaptest2");
  }
  close(fd1);
  unlink( "/tmp/mmaptest1");
}
   /*
    *  Unmap two pages.
    */
```

```
  if ( munmap(address, 2*PageSize) <    0) {
     perror("munmap() error");
  }
  else;
}
```

Output:

```
Data for file 1
Data for file 2
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

mmap64()—Memory map a Stream File (Large File Enabled)

Syntax

```
#include <sys/mman.h>

void *mmap64( void *addr,
              size_t len,
              int protection,
              int flags,
              int fildes,
              off64_t off);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Yes

The **mmap64()** function, similar to the **mmap()** function, is used to establish a memory mapping of a file.

For a discussion of the parameters, authorities required, return values, related information, and examples for **mmap()**, see "mmap()—Memory Map a File" on page 179—Memory Map a File.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs normally are hidden. To use the **mmap64()** API, you must compile the source with the _LARGE_FILE_API macro defined.
2. All of the usage notes for **mmap()** apply to **mmap64()**.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

mprotect()—Change Access Protection for Memory Mapping

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int  mprotect( void *addr,
               size_t  len,
               int     protection);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Yes

The **mprotect()** function is used to change the access protection of a memory mapping to that specified by *protection*. All whole pages of the process's address space, that were established by the **mmap()** function, addressed from *addr* continuing for a length of *len* will be affected by the change of access protection. You may specify PROT_NONE, PROT_READ, PROT_WRITE, or the inclusive or of PROT_READ and PROT_WRITE as values for the *protect* parameter.

Parameters

addr (Input) The starting address of the memory region for which the access is to be changed.

The *addr* argument must be a multiple of the page size. The "sysconf()—Get System Configuration Variables" on page 488 function may be used to determine the system page size.

len (Input) The length in bytes of the address range.

protection
(Input) The desired access protection. You may specify PROT_NONE, PROT_READ, PROT_WRITE, or the inclusive or of PROT_READ AND PROT_WRITE as values for the *protection* argument.

No access through the memory mapping will be permitted if PROT_NONE is specified.

Storage associated with the mapping cannot be altered unless the PROT_WRITE value is specified.

For shared mappings, PROT_WRITE requires that the file descriptor used to establish the map had been opened for write access. A shared mapping is a mapping created with the ***MAP_SHARED*** value of the flag parameter of the **mmap()** function.

Since private mappings do not alter the underlying file, PROT_WRITE may be specified for a mapping that had been created MAP_PRIVATE and had been opened for read access.

The following table shows the symbolic constants allowed for the protection argument.

Symbolic Constant	Decimal Value	Description
PROT_WRITE	2	Write access allowed.
PROT_READ	2	Read access allowed.
PROT_NONE	8	No access allowed.

Authorities

No authorization is required.

Return Value

Upon successful completion, the **mprotect()** function returns 0. Upon failure, -1 is returned and errno is set to the appropriate error number.

Error Conditions

When the **mprotect()** function fails, it returns -1 and sets there errno variable as follows.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems. The *protection* argument specifies a protection that violates the access permission the process has to the underlying mapped file. If the QSHRMEMCTL system value was 0 at the time the mapping was created, then this continues to limit the allowed access until the mapping is destroyed. An attempt to change the protection of a shared mapping to PROT_WRITE when the QSHRMEMCTL system value had been zero at the time of map creation will result in an errno of EACCES.
[EINVAL (page 540)]	For example, the *addr* argument is not a multiple of the page size. This error number also may indicate that the value of the *len* argument is 0.
[ENOMEM (page 543)]	The *addr* argument is out of the allowed range.
[ENOTAVAIL (page 547)]	
[ENOTSUP (page 542)]	For **mprotect()** this can be caused by an invalid combination of access requests on the *protection* parameter.

Error Messages

The following messages may be sent from this function.

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. The address pointer that was returned by **mmap()** can only be used with the V4R4M0 or later versions of the following languages:
   - ILE COBOL
   - ILE RPG
   - ILE C if the TERASPACE parameter is used when compiling the program.
2. » The application cannot write or store any data via the memory mapping which includes any tagged (16-byte) pointers because the pointer attribute will be lost. Some examples of tagged pointers include space pointers, system pointers, invocation pointers etc..

   If the DTAMDL(*LLP64) parameter is used when compiling an ILE C program, this limitation does not apply as the pointers will be 8 byte pointers, and their pointer attribute will be preserved. «

Related Information

- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "creat64()—Create or Rewrite a File (Large File Enabled)" on page 46—Create or Rewrite a File (Large File Enabled)
- "mmap()—Memory Map a File" on page 179—Memory Map a Stream File
- "munmap()—Remove Memory Mapping" on page 193—Remove Memory Mapping

- "msync()—Synchronize Modified Data with Mapped File" on page 190—Synchronize Modified Data with Mapped File
- "open()—Open File" on page 195—Open File
- "open64()—Open File (Large File Enabled)" on page 211—Open File (Large File Enabled)
- "sysconf()—Get System Configuration Variables" on page 488—Get system configuration variables

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a file, produces a memory mapping of the file using **mmap()**, and then changes the protection of the file using **mprotect()**.

```
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/mman.h>

main(void) {

  size_t bytesWritten =0;
  int fd;
  int PageSize;
  char   textÝ = "This is a test";

 if ( (PageSize = sysconf(_SC_PAGE_SIZE)) < 0) {
    perror("sysconf() Error=");
    return -1;
 }

 fd = open("/tmp/mmprotectTest",
             (O_CREAT | O_TRUNC | O_RDWR),
             (S_IRWXU | S_IRWXG | S_IRWXO) );
  if ( fd < 0 ) {
    perror("open() error");
    return fd;
  }

  off_t lastoffset = lseek( fd, 0, SEEK_SET);
  bytesWritten = write(fd, text, strlen(text));
  if (bytesWritten != strlen(text) ) {
    perror("write error. ");
    return -1;
  }

  lastoffset = lseek( fd, PageSize-1, SEEK_SET);
  bytesWritten = write(fd, " ", 1);   /* grow file to 1 page. */
  if ( bytesWritten != 1 ) {
     perror("write error. ");
     return -1;
  }
      /* mmap the file. */
  void *address;
  int len;
   off_t my_offset = 0;
   len = PageSize;   /* Map one page */
   address =
       mmap(NULL, len, PROT_NONE, MAP_SHARED, fd, my_offset);

   if ( address == MAP_FAILED ) {
       perror("mmap error. " );
```

```
        return -1;
      }

      if ( mprotect( address, len, PROT_WRITE) < 0 ) {
           perror("mprotect failed with error:");
           return -1;
       }
       else (void) printf("%s",address);

     close(fd);
     unlink("/tmp/mmprotectTest");
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

msync()—Synchronize Modified Data with Mapped File

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int  msync( void *addr,
            size_t  len,
            int     flags );
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Yes

The **msync()** function can be used to write modified data from a shared mapping (created using the **mmap()** function) to non-volatile storage or invalidate privately mapped pages. The data located through mapping address *addr* for a length of *len* are either written to disk, or invalidated, depending on the value of *flags* and the private or shared nature of the mapping.

Parameters

addr
The starting address of the memory region to be synchronized to permanent storage. The specified address must be a multiple of the page size.

len
The number of bytes affected. The length must not be zero. If the length is not a multiple of the page size the system will round this value to the next page boundary.

flags
The desired synchronization.

The following table shows the symbolic constants allowed for the flags parameter.

Symbolic Constant	Decimal Value	Description
MS_ASYNC	1	Perform asynchronous writes.
MS_SYNC	2	Perform synchronous writes.
MS_INVALIDATE	4	Invalidate privately cached data

The MS_SYNC and MS_ASYNC options are mutually exclusive. The MS_SYNC and MS_ASYNC options are ignored if the memory map was created with the MAP_PRIVATE option.

The MS_INVALIDATE option is used to discard changes made to a memory map created with the MAP_PRIVATE option. The private memory map is synchronized with the current data in the

file. Any reference subsequent to the execution of the **msync()** function that invalidates a page will result in a reference to the current value of the file. The first modification of a page after the privately mapped page is invalidated results in the creation of a fresh private copy of that page. Subsequent modifications of this page prior to the next execution of an msync that invalidates the page will result in modifications to the same private copy of the page.

The MS_INVALIDATE value is ignored if the memory map was created with the MAP_SHARED option.

Authorities

No authorization is required.

Return Value

Upon successful completion, the **msync()** function returns 0.

Error Conditions

When the **msync()** function fails, it returns -1 and sets errno as follows.

Error condition	Additional information
[EINVAL (page 540)]	For example, the value of the *len* parameter may be zero. Or, the value of the *addr* may not be a multiple of the page size or is out of the allowed range.
[ENOTAVAIL (page 547)]	
[EUNKNOWN (page 544)]	

Error Messages

The following messages may be sent from this function.

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. The **msync()** function must be used to write changed pages of a shared mapping to disk. If a system crash occurs before the **msync()** function completes, some data may not be preserved.

   Process termination does not automatically write changed pages to disk. Some or all pages may be eventually written by the paging subsystem, but no guarantee is given. Therefore, if the data must be preserved the **msync()** function must be used to ensure changes made through a shared memory map are written to disk.
2. The address pointer that was returned by **mmap()** can only be used with the V4R4M0 or later versions of the following languages:
   - ILE COBOL
   - ILE RPG
   - ILE C if the TERASPACE parameter is used when compiling the program.
3. The application cannot write or store any data via the memory mapping which includes any tagged (16-byte) pointers because the pointer attribute will be lost. Some examples of tagged pointers include space pointers, system pointers, invocation pointers etc..

   If the DTAMDL(*LLP64) parameter is used when compiling an ILE C program, this limitation does not apply as the pointers will be 8 byte pointers, and their pointer attribute will be preserved.

Related Information

- "open()—Open File" on page 195—Open File
- "open64()—Open File (Large File Enabled)" on page 211—Open File (Large File Enabled)
- "mmap()—Memory Map a File" on page 179—Memory Map a Stream File
- "munmap()—Remove Memory Mapping" on page 193—Remove Memory Mapping
- "mprotect()—Change Access Protection for Memory Mapping" on page 186—Change Access Protection for Memory Mapping

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a file, creates a memory map, stores data into the file, and writes the data to disk using the **msync()** function.

```
#include <errno.h >
#include <fcntl.h >
#include <unistd.h >
#include <stdio.h >
#include <stdlib.h >
#include <string.h >
#include <sys/types.h >
#include <sys/mman.h >

main(void) {

  size_t bytesWritten =0;
  int fd;
  int PageSize;
  const char   textÝ = "This is a test";

 if ( (PageSize = sysconf(_SC_PAGE_SIZE)) < 0) {
    perror("sysconf() Error=");
    return -1;
 }

 fd = open("/tmp/mmsyncTest",
             (O_CREAT | O_TRUNC | O_RDWR),
             (S_IRWXU | S_IRWXG | S_IRWXO) );
  if ( fd < 0 ) {
    perror("open() error");
    return fd;
  }

  off_t lastoffset = lseek( fd, PageSize, SEEK_SET);
  bytesWritten = write(fd, " ", 1 );
  if (bytesWritten != 1 ) {
    perror("write error. ");
    return -1;
  }


      /* mmap the file. */
  void *address;
  int len;
   off_t my_offset = 0;
   len = PageSize;   /* Map one page */
   address =
        mmap(NULL, len, PROT_WRITE, MAP_SHARED, fd, my_offset);

   if ( address == MAP_FAILED ) {
       perror("mmap error. " );
       return -1;
     }
```

```
      /* Move some data into the file using memory map. */
    (void) strcpy( (char*) address, text);
      /* use msync to write changes to disk. */
    if ( msync( address, PageSize , MS_SYNC ) < 0 ) {
        perror("msync failed with error:");
        return -1;
     }
     else (void) printf("%s","msync completed successfully.");

   close(fd);
   unlink("/tmp/msyncTest");
}
```

Output:

```
This is a test.
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

munmap()—Remove Memory Mapping

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int  munmap ( void *addr,
              size_t  len );
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Yes

The **munmap()** function removes addressability to a range of memory mapped pages of a process's address space. All pages starting with *addr* and continuing for a length of *len* bytes are removed.

The address range specified must begin on a page boundary. Portions of the specified address range which are not mapped, or were not established by the **mmap()** function, are not affected by the **munmap()** function.

If the mapping was created ***MAP_PRIVATE*** then any private altered pages are discarded and the system storage associated with the copies are returned to the system free space.

When the mapping is removed, the reference associated with the pages mapped over the file is removed. If the file has no references other than those due to memory mapping and the remaining memory mappings are removed by the **munmap()** function, then the file becomes unreferenced. If the file becomes unreferenced due to an **munmap()** function call and the file is no longer linked, then the file will be deleted.

Parameters

addr The starting address of the memory region being removed.

The *addr* parameter must be a multiple of the page size. The value zero or NULL is not a valid starting address. The **sysconf()** function may be used to determine the system page size.

len (Input) The length of the address range. All whole pages beginning with *addr* for a length of *len* are included in the address range.

Authorities

No authorization is required.

Return Value

Upon successful completion, the **munmap()** function returns 0. Upon failure, -1 is returned and errno is set to the appropriate error number.

Error Conditions

When the **munmap()** function fails, it returns -1 and sets errno as follows.

Error condition	Additional information
[EINVAL (page 540)]	For example, for **munmap()** this may mean that the address range from *addr* and continuing for a length of *len* is outside the valid range allowed for a process. This error may also indicate that the value for the *addr* parameter is not a multiple of the page size. A value of 0 for parameter *len* also will result in this error number.
[ENOTAVAIL (page 547)]	
[EUNKNOWN (page 544)]	

Error Messages

The following messages may be sent from this function.

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. The address pointer that was returned by **mmap()** can only be used with the V4R4M0 or later versions of the following languages:
   - ILE COBOL
   - ILE RPG
   - ILE C if the TERASPACE parameter is used when compiling the program.
2. » The application cannot write or store any data via the memory mapping which includes any tagged (16-byte) pointers because the pointer attribute will be lost. Some examples of tagged pointers include space pointers, system pointers, invocation pointers etc..

   If the DTAMDL(*LLP64) parameter is used when compiling an ILE C program, this limitation does not apply as the pointers will be 8 byte pointers, and their pointer attribute will be preserved. «

Related Information

- "open()—Open File" on page 195—Open File
- "open64()—Open File (Large File Enabled)" on page 211—Open File (Large File Enabled)
- "mmap()—Memory Map a File" on page 179—Memory Map a Stream File
- "mprotect()—Change Access Protection for Memory Mapping" on page 186—Change Access Protection for Memory Mapping
- "msync()—Synchronize Modified Data with Mapped File" on page 190—Synchronize Modified Data with Mapped File

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a file, produces a memory mapping of the file using mmap(), and then removes the mapping using the **munmap()** function.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/mman.h>

main() {
  char fn[]="creat.file";
  char  text[]="This is a test";
  int fd;
  int PageSize;

  if ((fd =
       open(fn, O_CREAT | O_RDWR | O_APPEND,S_IRWXU)  < 0)
    perror("open() error");
  else if (write(fd, text, strlen(text)) < 0;
       error("write() error=");
    else if ( (PageSize=sysconf(_SC_PAGESIZE)) < 0 )
         error("sysconf() Error=");
    else {
    off_t lastoffset = lseek( fd, PageSize-1, SEEK_SET);
    write(fd, " ", 1);   /* grow file to 1 page. */
      /* mmap the file. */
    void *address;
    int len;
     my_offset = 0;

     len = 4096;   /* Map one page */
     address =
         mmap(NULL, len, PROT_READ, MAP_SHARED, fd, my_offset)
     if ( address != MAP_FAILED ) {
       if ( munmap( address, len ) ) == -1) {
          error("munmap failed with error:");
       }
     }

    close(fd);
    unlink(fn);
  }
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

---

open()—Open File

Syntax

```
#include <fcntl.h>

int open(const char *path, int oflag, . . .);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 205.

The **open()** function opens a file and returns a number called a **file descriptor**. You can use this file descriptor to refer to the file in subsequent I/O operations such as **read()** or **write()**. In these subsequent operations, the file descriptor is commonly identified by the argument *fildes* or *descriptor*. Each file opened by a job gets a new file descriptor.

If the last element of the *path* is a symbolic link, the **open()** function resolves the contents of the symbolic link.

open() positions the **file offset** (an indicator showing where the next read or write will take place in the file) at the beginning of the file. However, there are options that can change the position.

open() clears the FD_CLOEXEC file descriptor flag for the new file descriptor. Refer to "fcntl()—Perform File Control Command" on page 82 for additional information about the FD_CLOEXEC flag.

The **open()** function also can be used to open a directory. The resulting file descriptor can be used in some functions that have a *fildes* parameter.

If the file being opened has been saved and its storage freed, the file is restored during this **open()** function. The storage extension exit program registered against the QIBM_QTA_STOR_EX400 exit point is called to restore the object. (See the Storage Extension Exit Program for details). If the file cannot successfully be restored, **open()** fails with the EOFFLINE error number.

For information about the exit point which can be associated with **open()**, see "Integrated File System Scan on Open Exit Program" on page 523.

Parameters

path
: (Input) A pointer to the null-terminated path name of the file to be opened.

  This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

  When a new file is created, the new file name is assumed to be represented in the language and country currently in effect for the job.

  See "QlgOpen()—Open a File (using NLS-enabled path name)" on page 273 for a description and an example of supplying the *path* in any CCSID.

oflag
: (Input) The file status flags and file access modes of the file to be opened. See "Using the oflag Parameter" on page 197.

  Note: The **open64()** API sets the O_LARGEFILE flag internally.

mode
: (Input) An optional third parameter of type `mode_t` that is required if the O_CREAT flag is set. It specifies the file permission bits to be used when a file is created. For a description of the permission bits, see "chmod()—Change File Authorizations" on page 22.

conversion ID
: (Input) An optional fourth parameter of type `unsigned int` that is required if the O_CCSID or O_CODEPAGE flag is set.

  If the O_CCSID flag is set, this parameter specifies a CCSID. If the O_CODEPAGE flag is set, this parameter specifies a code page used to derive a CCSID.

  The specified or derived CCSID is assumed to be the CCSID of the data in the file, when a new file is created. This CCSID is associated with the file during file creation.

  When the O_TEXT_CREAT flag and its prerequisite flags are not set, the specified or derived CCSID is the CCSID in which data is to be returned (when reading from a file), or the CCSID in which data is being supplied (when writing to a file).

See “Using CCSIDs and code pages” on page 201 for more details.

text file creation conversion ID

(Input) An optional fifth parameter of type `unsigned int` that is required if the O_TEXT_CREAT flag, along with prerequisite flags O_TEXTDATA, O_CREAT, and either O_CCSID or O_CODEPAGE, is set. Note: because O_EXCL is not required, this parameter may apply to files that already exist.

When O_CCSID flag is set, this parameter specifies a CCSID. If the O_CODEPAGE flag is set, this parameter specifies a code page used to derive a CCSID.

The specified or derived CCSID will be used as the CCSID of this open instance. Therefore, this will be the CCSID in which data is to be returned (when reading from a file), or the CCSID in which data is being supplied (when writing to a file). Data will be stored in the CCSID associated with the open file. Note: if the file was not created by this open operation, the file’s CCSID may be different than the CCSID associated with the *conversion ID* parameter.

See “Using CCSIDs and code pages” on page 201 for more details.

Using the oflag Parameter

One of the following values *must* be specified in *oflag*:

O_RDONLY

Open for reading only.

O_WRONLY

Open for writing only.

O_RDWR

Open for both reading and writing.

One or more of the following also can be specified in *oflag*:

O_APPEND

Position the file offset at the end of the file before each write operation.

O_CREAT

The call to **open()** has a *mode* argument.

If the file being opened already exists, O_CREAT has no effect, except when O_EXCL is also specified (see the following description of O_EXCL.

If the file being opened does not exist, it is created » and then opened. Since the create and open operations occur as separate steps, error [EBUSY] could be received if another user opened the object with a conflicting file sharing mode after the create step, but before the open step. « The user ID (uid) of the file is set to the effective uid of the job. If the object is being created in the ″root″ (/), QOpenSys, and user-defined file systems, the following applies. If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the S_ISGID bit of the parent directory is on, the group ID (GID) of the new object is set to the GID of the parent directory. For all other file systems, the group ID (GID) of the file is set to the GID of the directory in which the file is created. File permission bits are set according to *mode*, except for those set in the file mode creation mask of the job. The S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits are also set according to *mode*. The file type bits in *mode* are ignored. All other bits in *mode* must be cleared (not set) or a [EINVAL] error is returned.

O_EXCL

Ignored if O_CREAT is not set. If both O_EXCL and O_CREAT are specified, **open()** fails if the file already exists. If both O_EXCL and O_CREAT are specified, and *path* names a symbolic link, **open()** fails regardless of the contents of the symbolic link.

O_LARGEFILE

Open a large file. The descriptor returned can be used with the other APIs to operate on files

larger than 2GB (GB = 1073741824) minus 1 byte. The file systems that do not support large files will just ignore the O_LARGEFILE open flag if it is set. The O_LARGEFILE flag is ignored by the file systems when **open()** is used to open a directory.

O_TRUNC
Truncate the file to zero length if the file exists and it is a "regular file" (a stream file that can support positioning the file offset). The mode and owner of the file are not changed. O_TRUNC applies only to regular files. O_TRUNC has no effect on FIFO special files. The O_TRUNC behavior applies only when the file is successfully opened with O_RDWR or O_WRONLY.

Truncation of the file will return the [EOVERFLOW] error if the file is larger than 2 GB minus 1 byte and if the O_LARGEFILE oflag is not also specified on the **open()** call. (Note that **open64()** sets the O_LARGEFILE oflag automatically.)

If the file exists and it is a regular file, the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode are cleared.

If the file has a digital signature, **open()** with the O_TRUNC oflag causes the signature to be deleted.

O_TEXTDATA
Determines how the data is processed when a file is opened.

- If O_TEXTDATA is specified, the data is processed as text.

  The data is read from the file and written to the file assuming it is in textual form. When the data is read from the file, it is converted from the CCSID of the file to the CCSID of the job or the CCSID specified by the application receiving the data. When data is written to the file, it is converted to the CCSID of the file from the CCSID of the job or the CCSID specified by the application.

  For true stream files, any line-formatting characters (such as carriage return, tab, and end-of-file) are just converted from one CCSID to another.

  When reading from a record file that is being used as a stream file, end-of-line characters are added to the end of the data in each record. When writing to the record file:

  - End-of-line characters are removed.
  - Records are padded with blanks (for a source physical file member) or nulls (for a data physical file member).
  - Tab characters are replaced by the appropriate number of blanks to the next tab position.
- If O_TEXTDATA is not specified, the data is processed as binary. The data is read from the file and written to the file without any conversion. The application is responsible for handling the data.

See "Using CCSIDs and code pages" on page 201 for more details on text conversions.

O_CCSID
The call to open has a fourth argument (*conversion ID*), which is to be interpreted as a CCSID. Text conversions between any two CCSIDs supported by the **iconv()** API can be performed.

This flag cannot be specified with the O_CODEPAGE flag.

See "Using CCSIDs and code pages" on page 201 for more details.

O_CODEPAGE
The call to open has a fourth argument (*conversion ID*), which is to be interpreted as a code page. Only single-byte-to-single-byte or double-byte-to-double-byte text conversions are allowed.

This flag cannot be specified with the O_CCSID flag.

See "Using CCSIDs and code pages" on page 201 for more details.

O_TEXT_CREAT
The call to open has a fifth argument (*text file creation conversion ID*), which is to be interpreted as either a code page or CCSID, depending on whether the O_CODEPAGE or O_CCSID was set.

If the O_TEXT_CREAT flag is specified, all of the following flags must also be specified: O_CREAT, O_TEXTDATA, and either O_CODEPAGE or O_CCSID. If all of these prerequisite flags are not specified when O_TEXT_CREAT is specified, then the call to open will fail with error condition [EINVAL].

This flag indicates that the textual data read from or written to this file will be converted between the CCSID specified or derived from the *text file creation conversion ID* and the CCSID of the file. When data is read from the file, it is converted from the CCSID of the file to the CCSID specified or derived from the *text file creation conversion ID*. When data is written to the file, it is converted to the CCSID of the file from the CCSID specified or derived from the *text file creation conversion ID*.

See "Using CCSIDs and code pages" on page 201 for more details.

O_INHERITMODE
Create the file with the same data authorities as the parent directory that the file is created in. Any data authorities passed in the *mode* parameter are ignored. The mode parameter, however, must still be specified with a valid mode value. This flag is ignored if the O_CREAT flag is not set.

The "root" (/), QOpenSys, QSYS.LIB, independent ASP QSYS.LIB, and QDLS file systems support this flag on an **open()** with the O_CREAT flag set. The QOPT file system ignores this flag because files in this file system do not have data authorities.

O_NONBLOCK
Return without delay from certain operations on this open descriptor.

If O_NONBLOCK is specified when opening a FIFO:

- An **open()** for reading only or reading and writing access returns without delay.
- An **open()** for writing only returns an error if no job currently has the FIFO open for reading. The *errno* value will be ENXIO.

If O_NONBLOCK is not specified when opening a FIFO:

- An **open()** for reading only blocks the calling thread until another thread opens the FIFO for writing.
- An **open()** for writing only blocks the calling thread until another thread opens the FIFO for reading.
- An **open()** for reading and writing returns without delay.

The O_NONBLOCK open flag is ignored for all other object types.

O_SYNC
Updates to the file will be performed synchronously. All file data and file attributes relative to the I/O operation are written to permanent storage before the update operation returns. Update operations include, but are not limited to, the following: **ftruncate()**, **open()** with O_TRUNC, **write()**, and **fclear()**.

O_DSYNC
Updates to the file will be performed synchronously, but only the file data is written to permanent storage before the update operation returns. Update operations include, but are not limited to, the following: **ftruncate()**, **open()** with O_TRUNC, **write()**, and **fclear()**.

O_RSYNC
Read operations to the file will be performed synchronously. Pending update requests affecting the data to be read are written to permanent storage. This flag is used in combination with O_SYNC or O_DSYNC. When O_RSYNC and O_SYNC are set, all file data and file attributes are

written to permanent storage before the read operation returns. When O_RSYNC and O_DSYNC are set, all file data is written to permanent storage before the read operation returns.

O_FORCE_SCAN
One or more of the following conditions will be ignored when determining whether the integrated file system scan-related exit programs will be called:

- The Scan file systems control (QSCANFSCTL) system value specification of *FSVRONLY.
- The object was marked to not be scanned (e.g. scan attribute is *NO).
- The object was marked to be scanned only if the object changed (e.g. scan attribute is *CHGONLY).

For example, an object is opened that has a scan attribute of *YES, and the open request is not through the file servers when *FSVRONLY is specified. If O_FORCE_SCAN is specified on that open request, the object will be scanned if all the remaining conditions are met. Similarly, if an object that has a scan attribute of *NO or *CHGONLY is opened with O_FORCE_SCAN specified, the object will be scanned if all the remaining conditions are met. For a list of the remaining conditions and more information, see “Integrated File System Scan on Open Exit Program” on page 523 and “Integrated File System Scan on Close Exit Program” on page 513.

A file sharing mode may also be specified in the *oflag*. If none are specified, a default sharing mode of O_SHARE_RDWR is used. No more than one of the following may be specified:

O_SHARE_RDONLY
Share with readers only. Open the file only if both of the following are true:

- The file currently is not open for writing.
- The access intent does not conflict with the sharing mode of another open instance of this file.

Once opened with this sharing mode, any request to open this file for writing fails with the [EBUSY] error.

O_SHARE_WRONLY
Share with writers only. Open the file only if both of the following are true:

- The file is not currently open for reading.
- The access intent does not conflict with the sharing mode of another open instance of this file.

Once opened with this sharing mode, any request to open this file for reading fails with the [EBUSY] error.

O_SHARE_RDWR
Share with readers and writers. Open the file only if the access intent of this open does not conflict with the sharing mode of another open instance of this file.

O_SHARE_NONE
Share with neither readers nor writers. Open the file only if the file is not currently open. Once the file is opened with this sharing mode, any request to open this file for reading or writing fails with the [EBUSY] error.

All other bits in *oflag* must be cleared (not set).

Notes:

1. If O_WRONLY or O_RDWR is specified and the file is checked out by a user profile other than that of the current job, the **open()** fails with the [EBUSY] error.
2. If O_WRONLY or O_RDWR is specified and the file is marked ″read-only,″ the **open()** fails with the [EROOBJ] error.

3. If O_CREAT is specified and the file did not previously exist, a successful **open()** sets the access time, change time, modification time, and creation time for the new file. It also updates the change time and modification time of the directory that contains the new file (the parent directory of the new file).

   If O_TRUNC is specified and the file previously existed, a successful **open()** updates the change time and modification time for the file.
4. Sharing Files

   If a sharing mode is not specified in the *oflag* parameter, a default sharing mode of O_SHARE_RDWR is used. The **open()** may fail with the [EBUSY] error number if the file is already open with a sharing mode that conflicts with the access intent of this **open()** request.

   Directories may only be opened with a sharing mode of O_SHARE_RDWR. If any other sharing mode is specified, the **open()** fails with error number [EINVAL].

   For *CHRSF files, a sharing mode of O_SHARE_RDWR is used regardless of the sharing mode specified in the *oflag* parameter. The sharing mode specified in the *oflag* parameter is ignored.

   The following table shows when conflicts will occur:

File Sharing Conflicts

	Sharing Mode			
Access Intent	**Readers Only**	**Writers Only**	**Readers and Writers**	**No Others (Exclusive)**
O_RDONLY	OK	EBUSY	OK	EBUSY
O_WRONLY	EBUSY	OK	OK	EBUSY
O_RDWR	EBUSY	EBUSY	OK	EBUSY

Using CCSIDs and code pages

If the O_CCSID or O_CODEPAGE flag is specified, but O_CREAT is not, the *mode* parameter must be specified, but its value will be ignored.

The value of *conversion ID* must be less than 65536. The [EINVAL] error will be returned if it is not.

When a new file is created:

- *conversion ID* is used to derive a CCSID to be associated with the new file (the ″file CCSID″) and this open instance (the ″open CCSID″). If the file is to contain textual data, this CCSID is assumed to be the CCSID of the data, unless the O_TEXT_CREAT flag and its prerequisite flags were also specified.
- If neither O_CCSID nor O_CODEPAGE is specified, or if O_CCSID is specified and *conversion ID* is zero (0), the file CCSID is set to the CCSID of the job. If the job CCSID is 65535, the file CCSID is set to the default CCSID of the job.
- For this open instance, if the O_TEXT_CREAT flag and its prerequisite flags were not specified, the file CCSID and open CCSID are the same and no text conversion will take place on data written to or read from the file, whether O_TEXTDATA is specified or not. If you would like to associate the new file with the CCSID specified in *conversion ID*, but you would also like to have text conversion occur between the file's CCSID and a different CCSID, consider using the O_TEXT_CREAT flag and corresponding *text file creation conversion ID* parameter.
- The QSYS.LIB and independent ASP QSYS.LIB file systems cannot associate the derived CCSID with the database file member being created. Rather, the CCSID of the new member is the CCSID of the database file in which the member is being created. Data read or written during this open instance is converted from or to the CCSID of the database file.

When an existing file is opened and O_TEXTDATA is **not** specified:

- The value of *conversion ID* is ignored.

When an existing file is opened and O_TEXTDATA is specified:

- *conversion ID* is used to derive a CCSID to be associated with this open instance (the "open CCSID").
- If neither O_CCSID nor O_CODEPAGE is specified, or if O_CCSID is specified and *conversion ID* is zero (0), the open CCSID is set to the CCSID of the job. If the job CCSID is 65535, the open CCSID is set to the default CCSID of the job.
- The system will convert from the file CCSID to the open CCSID when reading data from the file, and convert from the open CCSID to the file CCSID when writing data to the file.
- If O_CCSID is not specified, and the file CCSID and open CCSID are not the same, and one of them is not strictly single-byte, **open()** will fail with errno set to [ECONVERT].

See “Examples” on page 209 for a sample program that creates a new file and then opens it for data conversion.

Authorities

Note: Adopted authority is not used.

Authorization Required for open() (excluding QSYS.LIB, independent ASP QSYS.LIB,and QDLS)		
Object Referred to	**Authority Required**	**errno**
Each directory in the path name preceding the object to be opened	*X	EACCES
Existing object when access mode is O_RDONLY	*R	EACCES
Existing object when access mode is O_WRONLY	*W	EACCES
Existing object when access mode is O_RDWR	*RW	EACCES
Existing object when O_TRUNC is specified	*W	EACCES
Parent directory of object to be created when object does not exist and O_CREAT is specified	*WX	EACCES

Authorization Required for open() in the QSYS.LIB and independent ASP QSYS.LIB File Systems		
Object Referred to	**Authority Required**	**errno**
Each directory in the path name preceding the object to be opened	*X	EACCES
Existing object when access mode is O_RDONLY	*R	EACCES
Existing object when access mode is O_WRONLY	*W	EACCES
Existing object when access mode is O_RDWR	*RW	EACCES
Existing object when object is a save file	*RWX	EACCES
Existing object when O_TRUNC is specified	*W	EACCES
Parent directory of object to be created when object does not exist and O_CREAT is specified	*OBJMGT or *OBJALTER	EACCES
Parent directory of object to be created when object does not exist and object type is *USRSPC or save file	*RX and *Add	EACCES
Parent directory of the parent directory of object to be created when object does not exist, O_CREAT is specified, and object being created is a physical file member	*ADD	EACCES

Authorization Required for open() in the QDLS File System		
Object Referred to	**Authority Required**	**errno**
Each directory in the path name preceding the object to be opened	*X	EACCES
Existing object when access mode is O_RDONLY	*R	EACCES
Existing object when access mode is O_WRONLY	*W	EACCES
Existing object when access mode is O_RDWR	*RW	EACCES
Existing object when O_TRUNC is specified	*W	EACCES
Parent directory of object to be created when object does not exist and O_CREAT is specified	*CHANGE	EACCES

Return Value

value **open()** was successful. The value returned is the file descriptor.

-1 **open()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **open()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	**Additional information**
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	The open sharing mode may conflict with another open of this file, or O_WRONLY or O_RDWR is specified and the file is checked out by another user. In the QSYS.LIB and independent ASP QSYS.LIB file systems, if the O_TEXTDATA flag was specified, the file may be already open in this job or another job where the O_TEXTDATA flag was not specified. Or if the O_TEXTDATA flag was not specified, the file may be already open in this job or another job where the O_TEXTDATA flag was specified.
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EDEADLK (page 543)]	
[EEXIST (page 543)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	

Error condition	Additional information
[EINVAL (page 540)]	For example • O_RDONLY and O_TRUNC were both specified. • More than one of O_RDONLY, O_WRONLY, or O_RDWR are set in *oflag*. • More than one of O_SHARE_RDONLY, O_SHARE_WRONLY, O_SHARE_RDWR, or O_SHARE_NONE are set in *oflag*. • Unused bits in *oflag* are set and should be cleared. • Unused bits in *mode* are set and should be cleared. • It is not valid to open this type of object. • O_CODEPAGE and O_CCSID were both specified.
[EIO (page 540)]	
[EISDIR (page 544)]	The path name given is a directory. Write access or O_TRUNC has been specified and is not valid for a directory.
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[EMFILE (page 543)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENFILE (page 543)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOSYS (page 544)]	
[ENOSYSRSC (page 545)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSUP (page 542)]	
[ENXIO (page 541)]	O_NONBLOCK and O_WRONLY open flags are specified, *path* refers to a FIFO, and no job has the FIFO open for reading.
[EOFFLINE (page 545)]	
[EOVERFLOW (page 546)]	The size of the specified file cannot be represented correctly in a variable of type off_t (the file is larger than 2GB minus 1 byte).
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESCANFAILURE (page 547)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETXTBSY (page 547)]	
[EUNKNOWN (page 544)]	

Additionally, if interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	

Error condition	Additional information
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error number [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. ″Root″ (/), QOpenSys, and User-Defined File System Differences

   The user who creates the file becomes its owner. The S_ISGID bit of the directory affects what the group ID (GID) is for objects that are created in the directory. If the S_ISGID bit of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the S_ISGID bit of the parent directory is on, the group ID is copied from the parent directory in which the file is created.

   When you do not specify O_INHERITMODE for the *oflag* parameter, the owner, primary group, and public object authorities (*OBJEXIST, *OBJMGT, *OBJALTER, and *OBJREF) are copied from the parent directory's owner, primary group, and public object authorities. This occurs even when the new file has a different owner than the parent directory. The owner, primary group, and public data

authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except for those permissions that are also set in the file mode creation mask). The new file does not have any private authorities or authorization list. It only has authorities for the owner, primary group, and public.

When you specify O_INHERITMODE for the *oflag* parameter, the owner, primary group, and public data and object authorities (*R, *W, *X, *OBJEXIST, *OBJMGT, *OBJALTER, and *OBJREF) are copied from the parent directory's owner, primary group, and public data and object authorities. In addition, the private authorities (if any) and authorization list (if any) are copied from the parent directory. If the new file has a different owner than the parent directory and the new file's owner has a private authority in the parent directory, that private authority is not copied from the parent directory. The authority for the owner of the new file is copied from the owner of the parent directory.

There are some restrictions when opening a FIFO for text conversion and the CCSIDs involved are not strictly single-byte:

- Opening a FIFO for reading or reading and writing is not allowed. The *errno* global variable is set to [ENOTSUP].
- Any conversion between CCSIDs that are not strictly single-byte must be done by an open instance that has write only access.

3. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   The following object types are allowed to be opened:

   - *MBR (physical file member)
     The only types of physical files supported when specifying the O_TEXTDATA flag are program-described physical files that contain a single field and source physical files that contain a single text field. Externally described physical files are supported for binary access only.
   - *LIB (library)
   - *FILE (physical file or save file)
   - *USRSPC (user space)

   When creating a member, the ownership, group profile, and authorities are all derived from the member's parent physical file. The input *mode* value is ignored.

   The group ID is obtained from the primary user profile, if a group profile exists.

   The primary group authorities specified in *mode* are not saved if no primary group exists.

   You cannot open a member in a file that has a mixed data CCSID.

   The file access time for a database member is updated using the normal rules that apply to database files. At most, the access time is updated once per day.

   Due to the restriction that only one job may have a database member open for writing at a time, the sharing modes O_SHARE_WRONLY and O_SHARE_RDWR do not provide the requested level of sharing.

   - If O_SHARE_WRONLY is specified, the **open()** succeeds. However, in all jobs other than the one that performed this **open()**, the actual enforced share mode for this file is equivalent to O_SHARE_NONE.
   - If O_SHARE_RDWR is specified, or if no share mode is specified, the **open()** succeeds. However, in all jobs other than the one that performed this **open()**, the actual enforced share mode is equivalent to O_SHARE_RDONLY.

   The **open()** of a database member fails with an [EBUSY] error under any of the following conditions:

   - The O_TEXTDATA flag is specified, but the file is already open in this job or another job where the O_TEXTDATA flag is not specified.
   - The O_TEXTDATA flag is not specified, but the file is already open in this job or another job where the O_TEXTDATA flag and write access are specified.
   - The O_TEXTDATA flag is specified and write access is requested, but the file is already open in this job or another job where O_TEXTDATA is specified and write access is also requested.

- The O_CREAT flag is specified, the member already exists, and the QSYS.LIB or independent ASP QSYS.LIB file system cannot get exclusive access to the member. They must have exclusive access to clear the old member.
- The O_TEXTDATA flag is not specified (binary mode) and more than one job tries to obtain write access to the member. This condition does not apply to PC clients. Because PC clients share the same server job, they can share access to the member.
- The user attempts to open a member with access intentions that conflict with existing object locks on the member.

This function will fail with error number [ENOTSAFE] if the object on which this function is operating is a save file and multiple threads exist in the job.

This function will fail with error number [ENOTSUP] if the file specified is a save file and the O_RDWR flag is specified. A save file can be opened for either reading only or writing only.

This function will fail with error number [ENOTSUP] if the file specified is a save file and the O_TEXTDATA flag is specified.

If a save file containing data is opened for writing, the O_APPEND or O_TRUNC flag must be specified. Otherwise, the **open()** will fail with error number [ENOTSUP].

There are some restrictions on sharing modes when opening a save file.

a. A save file may not have more than one open descriptor per job, regardless of the sharing mode specified.

   - A save file currently open for reading only cannot be opened again in the same job for reading or writing. The **open()** will fail with errno set to [EBUSY].
   - A save file currently open for writing only cannot be opened again in the same job for reading or writing. The **open()** will fail with errno set to [EBUSY].

b. Due to the restriction that only one job may have a save file open when the save file is open for writing, the sharing modes O_SHARE_WRONLY and O_SHARE_RDWR do not provide the reqested level of sharing.

   - If O_SHARE_WRONLY is specified, the **open()** succeeds. However, in all jobs other than the one that performed this **open()**, the actual enforced share mode for this file is equivalent to O_SHARE_NONE.
   - If O_SHARE_RDWR is specified and the file is opened for reading only, the **open()** succeeds. However, in all jobs other than the one that performed this **open()**, the actual enforced share mode is equivalent to O_SHARE_RDONLY.
   - If O_SHARE_RDWR is specified and the file is opened for writing only, the **open()** succeeds. However, in all jobs other than the one that performed this **open()**, the actual enforced share mode is equivalent to O_SHARE_NONE.

Note: Unpredictable results, including loss of data, could occur if, in the same job, a user tries to open the same file for writing at the same time by using both **open()** API for stream file access and a data management open API for record access.

4. QDLS File System Differences

   When O_CREAT is specified and a new file is created:

   - the owner's object authority is set to *OBJMGT + *OBJEXIST + *OBJALTER + *OBJREF.
   - The primary group and public object authority and all other authorities are copied from the directory (folder) in which the file is created.
   - The owner, primary group, and public data authority (including *OBJOPR) are derived from the permissions specified in *mode* (except those permissions that are also set in the file mode creation mask).

   The primary group authorities specified in *mode* are not saved if no primary group exists.

QDLS does not store the language ID and country ID with its files. When this information is requested (using the readdir() function), QDLS returns the language ID and country ID of the system.

5. QOPT File System Differences

   When the volume on which the file is being opened is formatted in Universal Disk Format (UDF):

   - The authorization that is checked for the object and preceding directories in the path name follows the rules described in Authorization Required for open() (page 202).
   - The volume authorization list is checked for *USE when the access mode is O_RDONLY. The volume authorization list is checked for *CHANGE when the access mode is O_RDWR or O_WRONLY.
   - The user who creates the file becomes its owner.
   - The group ID is copied from the parent directory in which the file is created.
   - The owner, primary group, and public data authorities (*R, *W, and *X) are derived from the permissions specified in the mode (except those permissions that are also set in the file mode creation mask).
   - When O_INHERITMODE is specified for the oflag parameter, the data authorities are copied from the parent directory.
   - The sharing modes O_SHARE_RDONLY, O_SHARE_WRONLY, and O_SHARE_RDWR do not provide the requested level of sharing when the access mode is O_RDWR or O_WRONLY. When the access mode is O_RDWR or O_WRONLY, the resulting sharing mode semantic will be equivalent to O_SHARE_NONE.
   - For newly created files, the same uppercase and lowercase forms in which the names are entered are preserved. No distinction is made between uppercase and lowercase when searching for names.
   - This function will fail with error number [EINVAL] if the O_SYNC, O_DSYNC, or O_RSYNC open flag is specified.

   When the volume on which the file is being opened is not formatted in Universal Disk Format (UDF):

   - No authorization checks are made on the object or preceding directories in the path name.
   - The volume authorization list is checked for *USE when the access mode is O_RDONLY. The volume authorization list is checked for *CHANGE when the access mode is O_RDWR or O_WRONLY.
   - QDFTOWN becomes the owner of the file.
   - No group ID is assigned to the file.
   - The permissions specified in the mode are ignored. The owner, primary group, and public data authorities are set to RWX.
   - For newly created files, names are created in uppercase. No distinction is made between uppercase and lowercase when searching for names.

6. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. The creation of a file may fail if permissions and other attributes that are stored locally by the Network File System are more restrictive than those at the server. A later attempt to create a file can succeed when the locally stored data has been refreshed. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) The creation can also succeed after the file system has been remounted.

   If you try to re-create a file that was recently deleted, the request may fail because data that was stored locally by the Network File System still has a record of the file's existence. The creation succeeds when the locally stored data has been updated.

   Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more

restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations.

The sharing modes O_SHARE_RDONLY, O_SHARE_WRONLY, and O_SHARE_NONE do not provide the requested level of sharing. If any one of these share modes is specified, the resulting share mode semantic will be equivalent to O_SHARE_RDWR.

7. QNetWare File System Differences

   The QNetWare file system does not fully support mode bits. See Netware on iSeries in the iSeries Information Center for more information.
8. This function will fail with the [EOVERFLOW] error if the specified file exists and its size is too large to be represented in a variable of type off_t (the file is larger than 2 GB minus 1 byte).
9. When you develop in C-based languages and an application is compiled with the _LARGE_FILES macro defined, the **open()** API will be mapped to a call to the **open64()** API.
10. Using this function on the /dev/null or /dev/zero character special file, the oflag values of O_CREAT and O_TRUNC have no effect.
11. The O_SYNC, O_DSYNC, and O_RSYNC open flags will not cause updates made to the file via mapped access to be written to permanent storage.

Related Information

- The <**fcntl.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "close()—Close File or Socket Descriptor" on page 34—Close File or Socket Descriptor
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "dup()—Duplicate Open File Descriptor" on page 55—Duplicate Open File Descriptor
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "Integrated File System Scan on Open Exit Program" on page 523
- "lseek()—Set File Read/Write Offset" on page 157—Set File Read/Write Offset
- "open64()—Open File (Large File Enabled)" on page 211—Open File (Large File Enabled)
- "QlgOpen()—Open a File (using NLS-enabled path name)" on page 273—Open a File (using NLS-enabled path name)
- "read()—Read from Descriptor" on page 437—Read from Descriptor
- "stat()—Get File Information" on page 468—Get File Information
- "umask()—Set Authorization Mask for Job" on page 491—Set Authorization Mask for Job
- "write()—Write to Descriptor" on page 502—Write to Descriptor

Examples

See Code disclaimer information for information pertaining to code examples.

The following example opens an output file for appending. Because no sharing mode is specified, the O_SHARE_RDWR sharing mode is used.

```
int fildes;
fildes = open("outfile",O_WRONLY | O_APPEND);
```

The following example creates a new file with read, write, and execute permissions for the user creating the file. If the file already exists, the **open()** fails. If the **open()** succeeds, the file is opened for sharing with readers only.

```
fildes = open("newfile",O_WRONLY|O_CREAT|O_EXCL|O_SHARE_RDONLY,S_IRWXU);
```

This example first creates an output file for with a specified CCSID. The file is then closed and opened again with data conversion. The **open()** function is called twice because no data conversion would have occurred when using the first open's descriptor on read or write operations, even if O_TEXTDATA had been specified on that open; however, the second open could be eliminated entirely by using O_TEXT_CREAT on the first open. This is demonstrated in the code example immediately following this example. In this example, EBCDIC data is written to the file and converted to ASCII.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int fd;
  int rc;
  char name[]="/test.dat";
  char data[]="abcdefghijk";
  int oflag1 = O_CREAT | O_RDWR | O_CCSID;
  int oflag2 = O_RDWR | O_TEXTDATA | O_CCSID;
  mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR;
  unsigned int file_ccsid = 819;
  unsigned int open_ccsid = 37;

  /*************************************************/
  /*  First create the file with the CCSID 819.     */
  /*************************************************/

  if ((fd=open(name,oflag1,mode,file_ccsid)) < 0)
  {
    perror("open() for create failed");
    return(0);
  }

  if (close(fd) < 0)
  {
    perror("close() failed.");
    return(0);
  }

  /*************************************************/
  /*  Now open the file so EBCDIC (CCSID 37) data    */
  /*  written will be converted to ASCII (CCSID 819).*/
  /*************************************************/

  if ((fd=open(name,oflag2,mode,open_ccsid)) < 0)
  {
    perror("open() with translation failed");
    return(0);
  }

  /*************************************************/
  /* Write some EBCDIC data.                        */
  /*************************************************/

  if (-1 == (rc=write(fd, data, strlen(data))))
  {
    perror("write failed");
    return(0);
  }

  if (0 != (rc=close(fd)))
  {
    perror("close failed");
    return(0);
  }
}
```

In this second example, EBCDIC data is written to the file and converted to ASCII. This will produce the same results as the first example, except that it did it by only using one open instead of two.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int fd;
  int rc;
  char name[]="/test.dat";
  char data[]="abcdefghijk";
  int oflag1 = O_CREAT | O_RDWR | O_CCSID | O_TEXTDATA | O_TEXT_CREAT | O_EXCL;
  mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR;
  unsigned int file_ccsid = 819;
  unsigned int open_ccsid = 37;

  /**************************************************/
  /*  First create the file with the CCSID 819, and  */
  /*  open it such that the data is converted       */
  /*  between the the open CCSID of 37 and the      */
  /*  file's CCSID of 819 when writing data to it.   */
  /**************************************************/

  if ((fd=open(name,oflag1,mode,file_ccsid,open_ccsid)) < 0)
  {
    perror("open() for create failed");
    return(0);
  }

  /**************************************************/
  /* Write some EBCDIC data.                        */
  /**************************************************/

  if (-1 == (rc=write(fd, data, strlen(data))))
  {
    perror("write failed");
    return(0);
  }

  /**************************************************/
  /* Close the file.                                */
  /**************************************************/
  if (0 != (rc=close(fd)))
  {
    perror("close failed");
    return(0);
  }
}
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

open64()—Open File (Large File Enabled)

Syntax

```
#include <fcntl.h>

int open64(const char *path, int oflag, . . .);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "open()—Open File" on page 195.

The **open64()** function, similar to the **open()** function, opens a file and returns a number called a file descriptor. **open64()** differs from **open()** in that it automatically opens the file with the O_LARGEFILE flag set. For a further description of the open flags, see "Using the oflag Parameter" on page 197 in the **open()** API.

For a discussion of the parameters, authorities required, return values, related information, and examples for the **open()** and **open64()** APIs, see "open()—Open File" on page 195.

See "QlgOpen64()—Open File (large file enabled and using NLS-enabled path name)" on page 274 for a description and an example of supplying the *path* in any CCSID.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **open64()** API, you must compile the source with the _LARGE_FILE_API macro defined.
2. All of the usage notes for **open()** apply to **open64()** and **QlgOpen64()**. See "Usage Notes" on page 205 in the **open()** API.

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

---

opendir()—Open Directory

Syntax

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dirname);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 215.

The **opendir()** function opens a directory so that it can be read with the **readdir()** function. The variable *dirname* is a string giving the name of the directory to open. If the last component of *dirname* is a symbolic link, **opendir()** follows the symbolic link. As a result, the directory that the symbolic link refers to is opened. The functions **readdir()**, **rewinddir()**, and **closedir()** can be called after a successful call to **opendir()**. The first **readdir()** call reads the first entry in the directory.

Names returned on calls to **readdir()** are returned in the CCSID (coded character set identifier) in effect for the current job at the time this **opendir()** function is called. If the CCSID of the job is 65535, the default CCSID of the job is used. See "QlgOpendir()—Open Directory (using NLS-enabled path name)" on page 275 for specifying a different CCSID.

Parameters

dirname
(Input) A pointer to the null-terminated path name of the directory to be opened.

This parameter is assumed to be represented in the CCSID currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgOpendir()—Open Directory (using NLS-enabled path name)" on page 275 for a description and an example of supplying the *dirname* in any CCSID.

Authorities

Note: Adopted authority is not used.

Authorization required for opendir()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be opened	*X	EACCES
The directory to be opened	*R	EACCES

Return Value

value
: **opendir()** was successful. The value returned is a pointer to a `DIR`, representing an open directory stream. This `DIR` describes the directory and is used in subsequent operations on the directory using the **readdir()**, **rewinddir()**, and **closedir()** functions.

NULL pointer
: **opendir()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **opendir()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EEXIST (page 543)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	

Error condition	Additional information
[EMFILE (page 543)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENFILE (page 543)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     – ″Root″ (/)
     – QOpenSys
     – User-defined
     – QNTC
     – QSYS.LIB
     – Independent ASP QSYS.LIB
     – QOPT
     – Network File System
     – QFileSvr.400
2. The **opendir()** function uses a file descriptor for each open directory. Applications are limited to opening no more than OPEN_MAX files and directories, and are subject to receiving the [EMFILE] and [ENFILE] errors when too many file descriptors are in use. See the **sysconf()** function for a description of OPEN_MAX.

   The file descriptor that is used by **opendir()** will not be inherited in a child process that is created by the **spawn()** or **spawnp()** API.
3. **opendir()** may allocate memory from the user's heap.
4. Files that are added to the directory after the first call to **readdir()** following an **opendir()** or **rewinddir()** may not be returned on calls to **readdir()**, and files that are removed may still be returned on calls to **readdir()**.
5. QDLS File System Differences

   QDLS updates the access time on **opendir()**.
6. QOPT File System Differences

   If the directory exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the directory and preceding directories in the path name follows the rules described in Authorization required for opendir() (page 213). If the directory exists on a volume formatted in some other media format, no authorization checks are made on the directory being opened and each directory in the path name. The volume authorization list is checked for *USE authority regardless of the volume media format.

Related Information

- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**dirent.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "closedir()—Close Directory" on page 37—Close Directory
- "QlgOpendir()—Open Directory (using NLS-enabled path name)" on page 275—Open Directory
- "readdir_r()—Read Directory Entry" on page 447—Read Directory Entry
- "readdir_r()—Read Directory Entry" on page 447—Read Directory Entry
- "readdir_r_ts64()—Read Directory Entry" on page 451—Read Directory Entry
- "rewinddir()—Reset Directory Stream to Beginning" on page 461—Reset Directory Stream to Beginning
- spawn()—Spawn Process
- spawnp()—Spawn Process with Path

- "sysconf()—Get System Configuration Variables" on page 488—Get system configuration variables

Example

See Code disclaimer information for information pertaining to code examples.

The following example opens a directory:

```
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>

void traverse(char *fn, int indent) {
  DIR *dir;
  struct dirent *entry;
  int count;
  char path[1025]; /*** EXTRA STORAGE MAY BE NEEDED ***/
  struct stat info;

  for (count=0; count<indent; count++) printf("  ");
  printf("%s\n", fn);

  if ((dir = opendir(fn)) == NULL)
    perror("opendir() error");
  else {
    while ((entry = readdir(dir)) != NULL) {
      if (entry->d_name[0] != '.') {
        strcpy(path, fn);
        strcat(path, "/");
        strcat(path, entry->d_name);
        if (stat(path, &info) != 0)
          fprintf(stderr, "stat() error on %s: %s\n", path,
                  strerror(errno));
        else if (S_ISDIR(info.st_mode))
               traverse(path, indent+1);
      }
    }
    closedir(dir);
  }
}

main() {
  puts("Directory structure:");
  traverse("/etc", 0);
}
```

Output:

```
Directory structure:
/etc
  /etc/samples
    /etc/samples/IBM
  /etc/IBM
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

pathconf()—Get Configurable Path Name Variables

Syntax

```
#include <unistd.h>

long pathconf(const char *path, int name);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 220.

The **pathconf()** function lets an application determine the value of a configuration variable (*name*) associated with a particular file or directory (*path*).

If the named file is a symbolic link, **pathconf()** resolves the symbolic link.

Parameters

path
(Input) A pointer to the null-terminated path name of the file for which the value of the configuration variable is requested.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the process. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See “QlgPathconf()—Get Configurable Path Name Variables (using NLS-enabled path name)” on page 278 for a description and an example of supplying the *path* in any CCSID.

name
(Input) The name of the configuration variable value requested.

The value of *name* can be any one of the following set of symbols defined in the <**unistd.h**> header file, each standing for a configuration variable:

_PC_LINK_MAX

Represents LINK_MAX, which indicates the maximum number of links the file can have. If *path* is a directory, **pathconf()** returns the maximum number of links that can be established to the directory itself.

_PC_MAX_CANON

Represents MAX_CANON, which indicates the maximum number of bytes in a terminal canonical input line.

_PC_MAX_INPUT

Represents MAX_INPUT, which indicates the minimum number of bytes for which space is available in a terminal input queue. This available space is the maximum number of bytes that a portable application can have the user enter before the application actually reads the input.

_PC_NAME_MAX

Represents NAME_MAX, which indicates the maximum number of bytes in a file name (not including any terminating null at the end if the file name is stored as a string). This symbol refers only to the file name itself; that is, the last component of the path name of the file. **pathconf()** returns the maximum length of file names, even when the path does not refer to a directory.

This value is the number of bytes allowed in the file name if it were encoded in the CCSID of the job. If the CCSID is mixed, this number is an estimate and may be larger than the actual allowable maximum.

_PC_PATH_MAX

Represents PATH_MAX, which indicates the maximum number of bytes in a complete path name (not including any terminating null at the end if the path name is stored as a string). **pathconf()** returns the maximum length of a relative path name relative to *path*, even when *path* does not refer to a directory.

This value is the number of bytes allowed in the path name if it were encoded in the CCSID of the job. If the CCSID is mixed, this number is an estimate and may be larger than the actual allowable maximum.

_PC_PIPE_BUF

Represents PIPE_BUF, which indicates the maximum number of bytes that can be written ″atomically″ to a pipe. If more than this number of bytes are written to a pipe, the operation may take more than one physical write operation and physical read operation to read the data on the other end of the pipe. If *path* is a FIFO special file, **pathconf()** returns the value for the file itself. If *path* is a directory, **pathconf()** returns the value for any FIFOs that exist or that can be created under the directory. If *path* is any other kind of file, an error of [EINVAL] is returned.

_PC_CHOWN_RESTRICTED

Represents _POSIX_CHOWN_RESTRICTED, as defined in the <**unistd.h**> header file. It restricts use of **chown()** to a job with appropriate privileges, and allows the group ID of a file to be changed only to the effective group ID of the job or to one of its supplementary group IDs. If *path* is a directory, **pathconf()** returns the value for any kind of file under the directory, but not for subdirectories of the directory.

_PC_NO_TRUNC

Represents _POSIX_NO_TRUNC, as defined in the <**unistd.h**> header file. It generates an error if a file name is longer than NAME_MAX. If *path* refers to a directory, the value returned by **pathconf()** applies to all files under that directory.

_PC_VDISABLE

Represents _POSIX_VDISABLE, as defined in the <**unistd.h**> header file. This symbol indicates that terminal special characters can be disabled using this character value, if it is defined.

_PC_THREAD_SAFE

This symbol is used to determine if the object represented by *path* resides in a threadsafe file system. **pathconf()** returns the value 1 if the file system is threadsafe and 0 if the file system is not threadsafe. **fpathconf()** will never fail with error code [ENOTSAFE] when called with _PC_THREAD_SAFE.

Authorities

Note: Adopted authority is not used.

Authorization required for pathconf()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	None	None

Return Value

value **pathconf()** was successful. The value of the variable requested in *name* is returned.

-1 One of the following has occurred:

- A particular variable has no limit (for example, _PC_PATH_MAX). The *errno* global variable is not changed.
- **pathconf()** was not successful. The *errno* is set.

Error Conditions

If **fpathconf()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	For example, *name* is not a valid configuration variable name, or the given variable cannot be associated with the specified file.
[EIO (page 540)]	
[EISDIR (page 544)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	

Error condition	**Additional information**
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. When this function is called with any configuration variable name except _PC_THREAD_SAFE, the following usage note applies:

   - This function will fail with error code [ENOTSAFE] when all the following conditions are true:

     – Where multiple threads exist in the job.
     – The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

       - "Root" (/)
       - QOpenSys
       - User-defined
       - QNTC
       - QSYS.LIB
       - Independent ASP QSYS.LIB
       - QOPT
       - Network File System
       - QFileSvr.400

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chown()—Change Owner and Group of File" on page 29—Change Owner and Group of File
- "fpathconf()—Get Configurable Path Name Variables by Descriptor" on page 92—Get Configurable Path Name Variables by Descriptor
- "QlgPathconf()—Get Configurable Path Name Variables (using NLS-enabled path name)" on page 278—Get Configurable Path Name Variables

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines the maximum number of bytes in a file name:

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
```

```
main() {
  long result;

  errno = 0;
  puts("examining NAME_MAX limit for root filesystem");
  if ((result = pathconf("/", _PC_NAME_MAX)) == -1)
    if (errno == 0)
      puts("There is no limit to NAME_MAX.");
    else perror("pathconf() error");
  else
    printf("NAME_MAX is %ld\n", result);
}
```

Output:

```
examining NAME_MAX limit for root filesystem
NAME_MAX is 255
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

pipe()—Create an Interprocess Channel

Syntax

```
#include <unistd.h>

int pipe(int fildes[2]);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Yes

The **pipe()** function creates a data pipe and places two file descriptors, one each into the arguments *fildes[0]* and *fildes[1]*, that refer to the open file descriptions for the read and write ends of the pipe, respectively. Their integer values will be the two lowest available at the time of the **pipe()** call. The O_NONBLOCK and FD_CLOEXEC flags will be clear on both descriptors. NOTE: these flags can, however, be set by the **fcntl()** function.

Data can be written to the file descriptor *fildes[1]* and read from file descriptor *fildes[0]*. A read on the file descriptor *fildes[0]* will access data written to the file descriptor *fildes[1]* on a first-in-first-out basis. File descriptor *fildes[0]* is open for reading only. File descriptor *fildes[1]* is open for writing only.

The **pipe()** function is often used with the **spawn()** function to allow the parent and child processes to send data to each other.

Upon successful completion, **pipe()** will update the access time, change time, and modification time of the pipe.

Parameters

fildes[2]
(Output) An integer array of size 2 that will receive the pipe descriptors.

Authorities

None.

Return Value

0 **pipe()** was successful.
-1 **pipe()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **pipe()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EFAULT (page 541)]	
[EMFILE (page 543)]	
[ENFILE (page 543)]	
[ENOMEM (page 543)]	
[EUNKNOWN (page 544)]	

Usage Notes

1. If this function is called by a thread executing one of the scan-related exit programs (or any of its created threads), the descriptors that are returned are scan descriptors. See "Integrated File System Scan on Open Exit Program" on page 523 and "Integrated File System Scan on Close Exit Program" on page 513 for more information. If a process is spawned, these scan descriptors are not inherited by the spawned process and therefore cannot be used in that spawned process. Therefore, in this case, the descriptors returned by **pipe()** function will only work within the same process.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**fcntl.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "fstat()—Get File Information by Descriptor" on page 95—Get File Information by Descriptor
- "Qp0zPipe()—Create Interprocess Channel with Sockets" on page 419—Create Interprocess Channel with Sockets
- "read()—Read from Descriptor" on page 437—Read from Descriptor
- spawn()—Spawn Process
- "write()—Write to Descriptor" on page 502—Write to Descriptor

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a pipe, writes 10 bytes of data to the pipe, and then reads those 10 bytes of data from the pipe.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

void main()
{
   int fildes[2];
   int rc;
   char writeData[10];
   char readData[10];
   int bytesWritten;
```

```
   int bytesRead;

   memset(writeData,'A',10);

   if (-1 == pipe(fildes))
   {
      perror("pipe error");
      return;
   }

   if (-1 == (bytesWritten = write(fildes[1],
                                   writeData,
                                   10)))
   {
      perror("write error");
   }
   else
   {
      printf("wrote %d bytes\n",bytesWritten);

      if (-1 == (bytesRead = read(fildes[0],
                                 readData,
                                 10)))
      {
         perror("read error");
      }
      else
      {
         printf("read %d bytes\n",bytesRead);
      }
   }

   close(fildes[0]);
   close(fildes[1]);

   return;
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

pread()—Read from Descriptor with Offset

Syntax

```
#include <unistd.h>

ssize_t pread(int file_descriptor,
            void *buf, size_t nbyte, off_t offset);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 225.

From the file indicated by *file_descriptor*, the **pread()** function reads *nbyte* bytes of input into the memory area indicated by *buf*. The *offset* value defines the starting position in the file and the file pointer position is not changed.

See "read()—Read from Descriptor" on page 437 for more information relating to reading from a descriptor.

In the QSYS.LIB and independent ASP QSYS.LIB file systems, the offset will be ignored for a member while in text mode.

Parameters

file_descriptor
(Input) The descriptor to be read.

buf (Output) A pointer to a buffer in which the bytes read are placed.

nbyte (Input) The number of bytes to be read.

offset (Input) The offset to the desired starting position in the file.

Authorities

No authorization is required.

Return Value

value	**pread()** was successful. The value returned is the number of bytes actually read and placed in *buf*. This number is less than or equal to *nbyte*. It is less than *nbyte* only if **pread()** reached the end of the file before reading the requested number of bytes. If **pread()** is reading a regular file and encounters a part of the file that has not been written (but before the end of the file), **pread()** places bytes containing zeros into *buf* in place of the unwritten bytes.
-1	**pread()** was not successful. The *errno* global variable is set to indicate the error. If the value of *nbyte* is greater than SSIZE_MAX, **pread()** sets *errno* to [EINVAL].

Error Conditions

If **pread()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	**Additional information**
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	The file resides in a file system that does not support large files, and the starting offset of the file exceeds 2GB minus 2 bytes. This will also occur if the *offset* value is less than 0.
[EIO (page 540)]	
[ENOMEM (page 543)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ENXIO (page 541)]	
[EOVERFLOW (page 546)]	The file is a regular file, *nbyte* is greater than 0, the starting offset is before the end-of-file, and the starting offset is greater than or equal to 2GB minus 2 bytes.
[ERESTART (page 547)]	
[ESPIPE (page 544)]	

Error condition	Additional information
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB

- QOPT
- Network File System
- QFileSvr.400

2. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   This function will fail with error code [ENOTSAFE] if the object specified is a save file and multiple threads exist in the job.

   This function will fail with error code [EIO] if the file specified is a save file and the file does not contain complete save file data.

   The file access time for a database member is updated using the normal rules that apply to database files. At most, the access time is updated once per day.

   If you previously used the integrated file system interface to manipulate a member that contains an end-of-file character, you should avoid using other interfaces (such as the Source Entry Utility or database reads and writes) to manipulate the member. If you use other interfaces after using the integrated file system interface, the end-of-file information will be lost.

3. QOPT File System Differences

   The file access time is not updated on a **pread()** operation.

   When reading from files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being read are ignored.

4. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

   Reading and writing to files with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the **fcntl()** API to get and release these locks.

5. QFileSvr.400 File System Differences

   The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.

6. For file systems that do not support large files, **pread()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **pread()** will return [EOVERFLOW] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.
7. Using this function successfully on the /dev/null or /dev/zero character special file results in a return value of zero. In addition, the access time for the file is updated.
8. If *file_descriptor* refers to a descriptor obtained using the **open()** API with O_TEXTDATA and O_CCSID specified, the file CCSID and open CCSID are not the same, and the converted data could expand or contract, then the *offset* value must be 0.
9. If *file_descriptor* refers to a character special file, the *offset* value is ignored.

Related Information

- The <**limits.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**unistd.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “creat()—Create or Rewrite File” on page 40—Create or Rewrite File
- “dup()—Duplicate Open File Descriptor” on page 55—Duplicate Open File Descriptor

- "dup2()—Duplicate Open File Descriptor to Another Descriptor" on page 58—Duplicate Open File Descriptor to Another Descriptor
- "fclear()—Write (Binary Zeros) to Descriptor" on page 77—Write (Binary Zeros) to Descriptor
- "fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)" on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "ioctl()—Perform I/O Control Request" on page 141—Perform I/O Control Request
- "lseek()—Set File Read/Write Offset" on page 157—Set File Read/Write Offset
- "open()—Open File" on page 195—Open File
- "pread64()—Read from Descriptor with Offset (large file enabled)" on page 228—Read from Descriptor with Offset (large file enabled)
- "pwrite()—Write to Descriptor with Offset" on page 229—Write to Descriptor with Offset
- "pwrite64()—Write to Descriptor with Offset (large file enabled)" on page 234—Write to Descriptor with Offset (large file enabled)
- "read()—Read from Descriptor" on page 437—Read from Descriptor
- "readv()—Read from Descriptor Using Multiple Buffers" on page 455—Read from Descriptor Using Multiple Buffers
- recv()—Receive Data
- recvfrom()—Receive Data
- recvmsg()—Receive Data or Descriptors or Both
- "write()—Write to Descriptor" on page 502—Write to Descriptor
- "writev()—Write to Descriptor Using Multiple Buffers" on page 509—Write to Descriptor Using Multiple Buffers

Example

See Code disclaimer information for information pertaining to code examples.

The following example opens a file and reads input:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

main() {
  int ret, file_descriptor;
  off_t off=5;
  char buf[]="Test text";

  if ((file_descriptor = creat("test.output", S_IWUSR))!= 0)
     perror("creat() error");
  else {
    if (-1==(rc=write(file_descriptor, buf, sizof(buf)-1)))
       perror("write() error");
    if (close(file_descriptor)!= 0)
       perror("close() error");
  }

  if ((file_descriptor = open("test.output", O_RDONLY)) < 0)
    perror("open() error");
  else {
    ret = pread(file_descriptor, buf, ((sizeof(buf)-1)-off), off);
    buf[ret] = 0x00;
    printf("block pread: \n<%s>\n", buf);
    if (close(file_descriptor)!= 0)
       perror("close() error");
```

```
  }
  if (unlink("test.output")!= 0)
     perror("unlink() error");
}
```

Output:

```
block pread:
<text>
```

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

pread64()—Read from Descriptor with Offset (large file enabled)

Syntax

```
#include <unistd.h>

ssize_t pread64(int file_descriptor,
            void *buf, size_t nbyte, off64_t offset);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes."

From the file indicated by *file_descriptor*, the **pread64()** function reads *nbyte* bytes of input into the memory area indicated by *buf*. The *offset* value defines the starting position in the file and the file pointer position is not changed.

pread64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:

- Using the **open64()** function (see "open64()—Open File (Large File Enabled)" on page 211).
- Using the **open()** function (see "open()—Open File" on page 195) with O_LARGEFILE set in the oflag parameter.

For additional information about parameters, authorities, and error conditions, see "pread()—Read from Descriptor with Offset" on page 223.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **pread64** API, you must compile the source with the _LARGE_FILE_API macro defined.
2. All of the usage notes for **pread()** apply to **pread64()**. See *Usage Notes* in the **pread** API.

Example

See Code disclaimer information for information pertaining to code examples.

The following example opens a file and reads input:

```
#define _LARGE_FILE_API
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

main() {
  int ret, file_descriptor;
  off64_t  off=5;
  char buf[]="Test text";
```

```
  if ((file_descriptor = creat64("test.output", S_IWUSR))!= 0)
     perror("creat64() error");
  else {
    if (-1==(rc=write(file_descriptor, buf, sizof(buf)-1)))
       perror("write() error");
    if (close(file_descriptor)!= 0)
       perror("close() error");
  }

  if ((file_descriptor = open64("test.output", O_RDONLY)) < 0)
    perror("open64() error");
  else {
    ret = pread64(file_descriptor, buf, ((sizeof(buf)-1)-off), off);
    buf[ret] = 0x00;
    printf("block pread64: \n<%s>\n", buf);
    if (close(file_descriptor)!= 0)
       perror("close() error");
  }
  if (unlink("test.output")!= 0)
     perror("unlink() error");
}
```

Output:

```
block pread64:
<text>
```

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

pwrite()—Write to Descriptor with Offset

Syntax

```
#include <unistd.h>

ssize_t pwrite
  (int file_descriptor, const void *buf,
   size_t nbyte, off_t offset);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 231.

The **pwrite()** function writes *nbyte* bytes from *buf* to the file associated with *file_descriptor*. The *offset* value defines the starting position in the file and the file pointer position is not changed.

See "write()—Write to Descriptor" on page 502 for more information relating to writing to a descriptor.

The *offset* will also be ignored if *file_descriptor* refers to a descriptor obtained using the **open()** function with O_APPEND specified.

Parameters

file_descriptor
(Input) The descriptor of the file to which the data is to be written.

buf (Input) A pointer to a buffer containing the data to be written.

nbyte (Input) The size in bytes of the data to be written.

offset (Input) The offset to the desired starting position in the file.

Authorities

No authorization is required.

Return Value

value	**pwrite()** was successful. The value returned is the number of bytes actually written. This number is less than or equal to *nbyte*.
-1	**pwrite()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **pwrite()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFBIG (page 545)]	The size of the object would exceed the system allowed maximum size or the process soft file size limit. The file is a regular file, *nbyte* is greater than 0, and the starting offset is greater than or equal to 2 GB minus 2 bytes.
[EINTR (page 541)]	
[EINVAL (page 540)]	The file system that the file resides in does not support large files, and the starting offset exceeds 2GB minus 2 bytes. This will also occur if the *offset* value is less than 0.
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ENXIO (page 541)]	
[ERESTART (page 547)]	
[ETRUNC (page 540)]	
[ESPIPE (page 544)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

For a physical file member opened in text mode, the *offset* must be equal to the current file position. Otherwise, this operation will fail with error code [EIO].

This function will fail with error code [ENOTSAFE] if the object on which this function is operating is a save file and multiple threads exist in the job.

If the file specified is a save file, only complete records will be written into the save file. A **pwrite()** request that does not provide enough data to completely fill a save file record will cause the partial record's data to be saved by the file system. The saved partial record will then be combined with additional data on subsequent **pwrite()**'s until a complete record may be written into the save file. If the save file is closed prior to a saved partial record being written into the save file, then the saved partial record is discarded, and the data in that partial record will need to be written again by the application.

A successful **pwrite()** updates the change, modification, and access times for a database member using the normal rules that apply to database files. At most, the access time is updated once per day.

You should be careful when writing end-of-file characters in the QSYS.LIB and independent ASP QSYS.LIB file systems. For these file systems, end-of-file characters are symbolic; that is, they are stored outside the file member. However, some situations can result in actual, nonsymbolic end-of-file characters being written to a member. These nonsymbolic end-of-file characters could cause some tools or utilities to fail. For example:

- If you previously wrote an end-of-file character as the last character of a member, do not continue to write data after that end-of-file character. Continuing to write data will cause a nonsymbolic end-of-file to be written. As a result, a compile of the member could fail.
- If you previously wrote an end-of-file character as the last character of a member, do not write other end-of-file characters preceding it in the file. This will cause a nonsymbolic end-of-file to be written. As a result, a compile of the member could fail.
- If you previously used the integrated file system interface to manipulate a member that contains an end-of-file character, avoid using other interfaces (such as the Source Entry Utility or database reads and writes) to manipulate the member. If you use other interfaces after using the integrated file system interface, the end-of-file information will be lost.

3. QOPT File System Differences

   The change and modification times of the file are updated when the file is closed.

   When writing to files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being written are ignored.

4. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations (several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data).

   Reading and writing to files with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the **fcntl()** API to get and release these locks.

5. QFileSvr.400 File System Differences

   The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.

6. For the file systems that do not support large files, **pwrite()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **pwrite()** will return [EFBIG] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.

7. Using this function successfully on the /dev/null or /dev/zero character special file results in a return value of the total number of bytes requested to be written. No data is written to the /dev/null or /dev/zero character special file. In addition, the change and modification times for the file are updated.
8. If the write exceeds the process soft file size limit, signal SIFXFSZ is issued.
9. If *file_descriptor* refers to a descriptor obtained using the **open()** function with O_TEXTDATA and O_CCSID specified, the file CCSID and open CCSID are not the same, and the converted data could expand or contract, then the *offset* value must be 0.
10. If *file_descriptor* refers to a character special file, the *offset* value is ignored.

Related Information

- The <**fcntl.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "dup()—Duplicate Open File Descriptor" on page 55—Duplicate Open File Descriptor
- "dup2()—Duplicate Open File Descriptor to Another Descriptor" on page 58—Duplicate Open File Descriptor to Another Descriptor
- "fclear()—Write (Binary Zeros) to Descriptor" on page 77—Write (Binary Zeros) to Descriptor
- "fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)" on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "ioctl()—Perform I/O Control Request" on page 141—Perform I/O Control Request
- "lseek()—Set File Read/Write Offset" on page 157—Set File Read/Write Offset
- "open()—Open File" on page 195—Open File
- "pread()—Read from Descriptor with Offset" on page 223—Read from Descriptor with Offset
- "pread64()—Read from Descriptor with Offset (large file enabled)" on page 228—Read from Descriptor with Offset (large file enabled)
- "pwrite64()—Write to Descriptor with Offset (large file enabled)" on page 234—Write to Descriptor with Offset (large file enabled)
- "read()—Read from Descriptor" on page 437—Read from Descriptor
- "readv()—Read from Descriptor Using Multiple Buffers" on page 455—Read from Descriptor Using Multiple Buffers
- send()—Send Data
- sendmsg()—Send Data or Descriptors or Both
- sendto()—Send Data
- "write()—Write to Descriptor" on page 502—Write to Descriptor
- "writev()—Write to Descriptor Using Multiple Buffers" on page 509—Write to Descriptor Using Multiple Buffers

Example

See Code disclaimer information for information pertaining to code examples.

The following example writes a specific number of bytes to a file:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define mega_string_len 1000000

main() {
  char *mega_string;
  int  file_descriptor;
  int  ret;
  off_t  off=5;
  char fn[]="write.file";

  if ((mega_string = (char*) malloc(mega_string_len+off)) == NULL)
    perror("malloc() error");
  else if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    memset(mega_string, '0', mega_string_len);
    if ((ret = pwrite(file_descriptor, mega_string, mega_string_len, off)) == -1)
      perror("pwrite() error");
    else printf("pwrite() wrote %d bytes at offset %d\n", ret, off);
    if (close(file_descriptor)!= 0)
       perror("close() error");
    if (unlink(fn)!= 0)
       perror("unlink() error");
  }
  free(mega_string);
}
```

Output:

```
pwrite() wrote 1000000 bytes at offset 5
```

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

pwrite64()—Write to Descriptor with Offset (large file enabled)

Syntax

```
#include <unistd.h>

ssize_t pwrite64
  (int file_descriptor, const void *buf,
   size_t nbyte, off64_t offset);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 235.

The **pwrite64()** function writes *nbyte* bytes from *buf* to the file associated with *file_descriptor*. The *offset* value defines the starting position in the file and the file pointer position is not changed.

The *offset* will also be ignored if *file_descriptor* refers to a descriptor obtained using the **open()** function with O_APPEND specified.

pwrite64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte as long as the file has been opened by either of the following:

- Using the **open64()** function (see "open64()—Open File (Large File Enabled)" on page 211).
- Using the **open()** function (see "open()—Open File" on page 195) with O_LARGEFILE set in the oflag parameter.

For additional information about parameters, authorities, and error conditions, see "pwrite()—Write to Descriptor with Offset" on page 229.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **pwrite64** API, you must compile the source with the _LARGE_FILE_API macro defined.
2. All of the usage notes for **pwrite()** apply to **pwrite64()**. See *Usage Notes* in the **pwrite** API.

Example

See Code disclaimer information for information pertaining to code examples.

The following example writes a specific number of bytes to a file:

```
#define _LARGE_FILE_API
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define mega_string_len 1000000

main() {
  char *mega_string;
  int  file_descriptor;
  int  ret;
  off64_t  off=5;
  char fn[]="write.file";

  if ((mega_string = (char*) malloc(mega_string_len+off)) == NULL)
    perror("malloc() error");
  else if ((file_descriptor = creat64(fn, S_IWUSR)) < 0)
    perror("creat64() error");
  else {
    memset(mega_string, '0', mega_string_len);
    if ((ret = pwrite64(file_descriptor, mega_string, mega_string_len, off)) == -1)
      perror("pwrite64() error");
    else printf("pwrite64() wrote %d bytes at offset %d\n", ret, off);
    if (close(file_descriptor)!= 0)
       perror("close() error");
    if (unlink(fn)!= 0)
       perror("unlink() error");
  }
  free(mega_string);
}
```

Output:

```
pwrite64() wrote 1000000 bytes at offset 5
```

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

QlgAccess()—Determine File Accessibility (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgAccess(const Qlg_Path_Name_T *path, int amode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for “access()—Determine File Accessibility” on page 10.

The **QlgAccess()** function, like the **access()** function, determines whether a file can be accessed in a particular manner. The difference is that the **QlgAccess()** function takes a pointer to a Qlg_Path_Name_T structure, while **access()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see “access()—Determine File Accessibility” on page 10—Determine File Accessibility.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name for the file to be checked for accessibility. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- “access()—Determine File Accessibility” on page 10—Determine File Accessiblity
- “accessx()—Determine File Accessibility for a Class of Users” on page 14—Determine File Accessibility for Class of Users
- “faccessx()—Determine File Accessibility for a Class of Users” on page 61—Determine File Accessibility for Class of Users
- “QlgAccessx()—Determine File Accessibility for a Class of Users (using NLS-enabled path name)” on page 237—Determine File Accessibility for Class of Users (using NLS-enabled path name)
- “QlgChmod()—Change File Authorizations (using NLS-enabled path name)” on page 240—Change File Authorizations (using NLS-enabled path name)
- “QlgStat()—Get File Information (using NLS-enabled path name)” on page 293—Get File Information (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines how a file is accessed:

```
#include <stdio.h>
#include <unistd.h>

main()
{

  /***************************************************************/
  /*   Defininitons                                              */
  /***************************************************************/
#define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this must */
                  /* be a pointer to the path name.  */
```

```
  };
  struct pnstruct path;

  /***************************************************************/
  /*   Initialize Qlg_Path_Name_T parameters                     */
  /***************************************************************/
  memset((void*)path name, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);


  if (QlgAccess((Qlg_Path_Name_T *)&path, F_OK) != 0)
    printf("'%s' does not exist!\n", mypath);
  else {
    if (QlgAccess((Qlg_Path_Name_T *)&path, R_OK) == 0)
      printf("You have read access to '%s'\n", mypath);
    if (QlgAccess((Qlg_Path_Name_T *)&path, W_OK) == 0)
      printf("You have write access to '%s'\n", mypath);
    if (QlgAccess((Qlg_Path_Name_T *)&path, X_OK) == 0)
      printf("You have search access to '%s'\n", mypath);
  }
}
```

Output:

The output from a user with read and execute access is:

```
You have read access to '/'
You have write access to '/'
You have search access to '/'
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

QlgAccessx()—Determine File Accessibility for a Class of Users (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgAccessx(const Qlg_Path_Name_T *path, int amode, int who);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "accessx()—Determine File Accessibility for a Class of Users" on page 14.

The **QlgAccessx()** function, like the **accessx()** function, determines whether a file can be accessed in a particular manner by a specified class of users. The difference is that the **QlgAccessx()** function takes a pointer to a Qlg_Path_Name_T structure, while **accessx()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "accessx()—Determine File Accessibility for a Class of Users" on page 14—Determine File Accessibility for a Class of Users.

Parameters

path
: (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name for the file to be checked for accessibility. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "access()—Determine File Accessibility" on page 10—Determine File Accessiblity
- "accessx()—Determine File Accessibility for a Class of Users" on page 14—Determine File Accessibility for a Class of Users
- "faccessx()—Determine File Accessibility for a Class of Users" on page 61—Determine File Accessibility for a Class of Users
- "QlgAccess()—Determine File Accessibility (using NLS-enabled path name)" on page 235—Determine File Accessibility (using NLS-enabled path name)
- "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240—Change File Authorizations (using NLS-enabled path name)
- "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293—Get File Information (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines how a file is accessed:

```
#include <stdio.h>
#include <unistd.h>

main()
{

  /*****************************************************************/
  /*   Defininitons                                                */
  /*****************************************************************/
#define mypath "/myfile"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

  /*****************************************************************/
  /*   Initialize Qlg_Path_Name_T parameters                       */
  /*****************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if (QlgAccessx((Qlg_Path_Name_T *)&path, R_OK, ACC_OTHERS) == 0)
      printf("Someone besides the owner has read access to '%s'\n", mypath);
  if (QlgAccessx((Qlg_Path_Name_T *)&path, W_OK, ACC_OTHERS) == 0)
```

```
      printf("Someone besides the owner has write access to '%s'\n", mypath);
  if (QlgAccessx((Qlg_Path_Name_T *)&path, X_OK, ACC_OTHERS) == 0)
      printf("Someone besides the owner has search access to '%s'\n", mypath);
}
```

Output:

In this example **QlgAccessx()** was called on '/myfile'. The following would be the output if someone other than the owner has *R authority, someone besides the owner has *W authority, and noone other than the owner has *X authority.

```
Someone besides the owner has read access to '/'
Someone besides the owner has write access to '/'
```

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

QlgChdir()—Change Current Directory (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgChdir(const Qlg_Path_Name_T *path);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "chdir()—Change Current Directory" on page 19.

The **QlgChdir()** function, like the **chdir()** function, makes the directory named by *path* the new current directory. The difference is that the **QlgChdir()** function takes a pointer to a Qlg_Path_Name_T structure, while **chdir()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "chdir()—Change Current Directory" on page 19—Change Current Directory.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the directory that should become the current directory. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "chdir()—Change Current Directory" on page 19—Change Current Directory
- "QlgGetcwd()—Get Current Directory (using NLS-enabled path name)" on page 248—Get Current Directory (using NLS-enabled path name)
- "fchdir()—Change Current Directory by Descriptor" on page 66—Change Current Directory by Descriptor

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **QlgChdir()**:

```
#include <stdio.h>
#include <unistd.h>

main() {
#define mypath "/tmpXXX"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this */
                  /* this be a pointer to the path name.  */
  };
  struct pnstruct path;

  /***************************************************************/
  /*   Initialize Qlg_Path_Name_T parameters                     */
  /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if (QlgChdir((Qlg_Path_Name_T *)&path) != 0)
  {
    printf("QlgChdir() to /tmpXXX failed.");
  }
  else
  {
   printf("QlgChdir() changed the current directory ");
   printf("to '%s'.\n", mypath);
  }
}
```

Output:

```
QlgChdir() changed the current directory to '/tmpxxx'.
(or if error, such as path not found: QlgChdir() to /tmpXXX failed.)
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgChmod()—Change File Authorizations (using NLS-enabled path name)

Syntax

```
#include <sys/stat.h>

int QlgChmod(Qlg_Path_Name_T *path, mode_t mode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "chmod()—Change File Authorizations" on page 22.

The **QlgChmod()** function, like the **chmod()** function, changes S_ISUID, S_ISGID, S_ISVTX, and the permission bits of the file or directory specified in *path* to the corresponding bits specified in *mode*. The difference is that the **QlgChmod()** function takes a pointer to a Qlg_Path_Name_T structure, while **chmod()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see “chmod()—Change File Authorizations” on page 22—Change File Authorizations.

Parameters

path
: (Input) A pointer to a Qlg_Path_Name_T structure that contains the path name or a pointer to the path name of the file whose mode is being changed. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- “chmod()—Change File Authorizations” on page 22—Change File Authorizations
- “QlgChown()—Change Owner and Group of File (using NLS-enabled path name)” on page 242—Change Owner and Group of File (using NLS-enabled path name)
- “QlgMkdir()—Make Directory (using NLS-enabled path name)” on page 270—Make Directory (using NLS-enabled path name)
- “QlgStat()—Get File Information (using NLS-enabled path name)” on page 293—Get File Information (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example changes the permissions for a file:

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <Qp0lstdi.h>

main() {
  int file_descriptor;
  struct stat info;

 #define mypath "temp.file"
 const char US_const[3]= "US";
 const char Language_const[4] ="ENU";
 typedef struct pnstruct
 {
   Qlg_Path_Name_T qlg_struct;
   char pn[100]; /* This array size must be >= the  */
                 /* length of the path name or this must */
                 /* be a pointer to the path name.  */
 };
 struct pnstruct path;

  /***************************************************************/
  /*   Initialize Qlg_Path_Name_T parameters                     */
  /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
```

```
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)&path, S_IWUSR)) == -1)
    perror("QlgCreat() error");
  else {
    close(file_descriptor);
    QlgStat((Qlg_Path_Name_T *)&path, &info);
    printf("original permissions were: %08o\n", info.st_mode);
    if (QlgChmod((Qlg_Path_Name_T *)&path, S_IRWXU|S_IRWXG) != 0)
      perror("QlgChmod() error");
    else {
      QlgStat((Qlg_Path_Name_T *)&path, &info);
      printf("after QlgChmod(), permissions are: %08o\n", info.st_mode);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path);
  }
}
```

Output:

```
original permissions were: 00100200
after QlgChmod(), permissions are: 00100770
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgChown()—Change Owner and Group of File (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgChown(Qlg_Path_Name_T *path, uid_t owner,gid_t
group);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "chown()—Change Owner and Group of File" on page 29.

The **QlgChown()** function, like the **chown()** function, changes the owner and group of a file. The difference is that the **QlgChown()** function takes a pointer to a Qlg_Path_Name_T structure, while **chown()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "chown()—Change Owner and Group of File" on page 29—Change Owner and Group of File.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file whose owner and group are being changed. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "chown()—Change Owner and Group of File" on page 29—Change Owner and Group of File

- "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240—Change File Authorizations (using NLS-enabled path name)
- "QlgLstat()—Get File or Link Information (using NLS-enabled path name)" on page 265—Get File or Link Information (using NLS-enabled path name)
- "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293—Get File Information (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example changes the owner and group of a file:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <Qp0lstdi.h>

main() {
  int file_descriptor;
  struct stat info;

 #define mypath "temp.file"
 const char US_const[3]= "US";
 const char Language_const[4] ="ENU";
 typedef struct pnstruct
 {
   Qlg_Path_Name_T qlg_struct;
   char pn[100]; /* This array size must be >= the  */
                 /* length of the path name or this must */
                 /* be a pointer to the path name.  */
 };
 struct pnstruct path;
   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);


  if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)&path, S_IRWXU)) == -1)
    perror("creat() error");
  else {
    close(file_descriptor);
    QlgStat((Qlg_Path_Name_T *)&path, &info);
    printf("original owner was %d and group was %d\n", info.st_uid,
           info.st_gid);
    if (QlgChown((Qlg_Path_Name_T *)&path, 152, 0) != 0)
      perror("QlgChown() error");
    else {
      QlgStat((Qlg_Path_Name_T *)&path, &info);
      printf("after QlgChown(), owner is %d and group is %d\n",
             info.st_uid, info.st_gid);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path);
  }
}
```

Output:

```
original owner was 137 and group was 0
after QlgChown(), owner is 152 and group is 0
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgCreat()—Create or Rewrite File (using NLS-enabled path name)

Syntax

```
#include <fcntl.h>

int QlgCreat(Qlg_Path_Name_T *path, mode_t mode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for “open()—Open File” on page 195.

The **QlgCreat()** function, like the **creat()** function, creates a new file or rewrites an existing file so that it is truncated to zero length. The difference is that the **QlgCreat()** function takes a pointer to a Qlg_Path_Name_T structure, while **creat()** takes a pointer to a character string. See “open()—Open File” on page 195—Open File for more details on how the function call

```
QlgCreat(path,mode);
```

is equivalent to the call

```
QlgOpen(path, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see “creat()—Create or Rewrite File” on page 40—Create or Rewrite File or “open()—Open File” on page 195—Open File.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file to be created or rewritten. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- “creat()—Create or Rewrite File” on page 40—Create or Rewrite File
- “QlgCreat64()—Create or Rewrite a File (large file enabled and using NLS-enabled path name)” on page 245—Create or Rewrite a File (large file enabled and using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a file:

```
#include <stdio.h>
#include <fcntl.h>
#include <Qp0lstdi.h>

main() {
  char text[]="This is a test";
  int file_descriptor;
```

```
#define mypath "creat.file"
 const char US_const[3]= "US";
 const char Language_const[4] ="ENU";
 typedef struct pnstruct
 {
   Qlg_Path_Name_T qlg_struct;
   char pn[100]; /* This array size must be >= the  */
                 /* length of the path name or this must */
                 /* be a pointer to the path name.  */
 };
 struct pnstruct path;

  /*****************************************************************/
  /*   Initialize Qlg_Path_Name_T parameters                       */
  /*****************************************************************/
 memset((void*)&path, 0x00, sizeof(struct pnstruct));
 path.qlg_struct.CCSID = 37;
 memcpy(path.qlg_struct.Country_ID,US_const,2);
 memcpy(path.qlg_struct.Language_ID,Language_const,3);
 path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
 path.qlg_struct.Path_Length = sizeof(mypath)-1;
 path.qlg_struct.Path_Name_Delimiter[0] = '/';
 memcpy(path.pn,mypath,sizeof(mypath)-1);
 if ((file_descriptor =
       QlgCreat((Qlg_Path_Name_T *)&path, S_IRUSR | S_IWUSR)) < 0)
   perror("QlgCreat() error");
 else {
   write(file_descriptor, text, strlen(text));
   close(file_descriptor);
   QlgUnlink((Qlg_Path_Name_T *)&path);
 }
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgCreat64()—Create or Rewrite a File (large file enabled and using NLS-enabled path name)

Syntax

```
#include <fcntl.h>

int QlgCreat64(Qlg_Path_Name_T *path,mode_t mode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "creat64()—Create or Rewrite a File (Large File Enabled)" on page 46.

The **QlgCreat64()** function, like the **creat64()** function, creates a new file or rewrites an existing file so that it is truncated to zero length. The difference is that the **QlgCreat64()** function takes a pointer to a Qlg_Path_Name_T structure, while **creat64()** takes a pointer to a character string. See "creat64()—Create or Rewrite a File (Large File Enabled)" on page 46—Create or Rewrite a File (Large File Enabled) and "open64()—Open File (Large File Enabled)" on page 211—Open File (Large File Enabled) for more details on how the function call

```
QlgCreat64(path,mode);
```

is equivalent to the call

```
QlgOpen64(path, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "creat64()—Create or Rewrite a File (Large File Enabled)" on page 46—Create or Rewrite a File (Large File Enabled) or "open64()—Open File (Large File Enabled)" on page 211—Open File (Large File Enabled).

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file to be created or rewritten. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "creat()—Create or Rewrite File" on page 40—Create or Rewrite a File
- "creat64()—Create or Rewrite a File (Large File Enabled)" on page 46—Create or Rewrite a File (Large File Enabled)

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a file:

```
#define _LARGE_FILE_API

#include <stdio.h>
#include <fcntl.h>
#include <Qp0lstdi.h>

main()
{
  char text[]="This is a test";
  int fd;
#define mypath "creat.file"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

  /*****************************************************************/
  /*   Initialize Qlg_Path_Name_T parameters                       */
  /*****************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if
   ((fd =
   QlgCreat64(
        (Qlg_Path_Name_T *)&path, S_IRUSR | S_IWUSR))
    < 0)
   {
```

```
    perror("QlgCreat64() error");
   }
  else {
    write(fd, text, strlen(text));
    close(fd);
    QlgUnlink((Qlg_Path_Name_T *)&path);
  }
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgCvtPathToQSYSObjName()— Resolve Integrated File System Path Name into QSYS Object Name (using NLS-enabled path name)

Syntax

```
#include <qp0lstdi.h>

void QlgCvtPathToQSYSObjName(
                              Qlg_Path_Name_T *path_name,
                              void            *qsys_info,
                              char             format_name[8],
                              uint             bytes_provided,
                              uint             desired_CCSID,
                              void            *error_code);
```

Service Program Name: QP0LLIB2
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "Qp0lCvtPathToQSYSObjName()— Resolve Integrated File System Path Name into QSYS Object Name" on page 308.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see "Qp0lCvtPathToQSYSObjName()— Resolve Integrated File System Path Name into QSYS Object Name" on page 308— Resolve Integrated File System Path Name into QSYS Object Name.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgGetAttr()—Get Attributes (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>
int QlgGetAttr
  (Qlg_Path_Name_T        *Path_Name,
   Qp0l_AttrTypes_List_t  *Attr_Array_ptr,
   char                   *Buffer_ptr,
   uint                    Buffer_Size_Provided,
   uint                   *Buffer_Size_Needed_ptr,
   uint                   *Num_Bytes_Returned_ptr,
   uint                    Follow_Symlnk, ...);
```

Service Program Name: QP0LLIB2
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "Qp0lGetAttr()—Get Attributes" on page 326.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, related information, and an example, see “Qp0lGetAttr()—Get Attributes” on page 326—Get Attributes.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgGetcwd()—Get Current Directory (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

 Qlg_Path_Name_T *QlgGetcwd(Qlg_Path_Name_T *buf,
 size_t size);
```

Service Program Name: QP0LLIB2
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for “getcwd()—Get Current Directory” on page 113.

The **QlgGetcwd()** function, like the **getcwd()** function, determines the absolute path name of the current directory and returns a pointer to it. The difference is that the pointer returned by **QlgGetcwd()** is a pointer to a Qlg_Path_Name_T structure that holds the absolute path name, while **getcwd()** returns a pointer to a character string or buffer that contains the null-terminated absolute path name.

Limited information on the *buf* parameter and on the *size* parameter is provided here. For more information on the parameters and for a discussion on authorities required, return values, and related information, see “getcwd()—Get Current Directory” on page 113—Get Current Directory.

Parameters

buf (Output) A pointer to a Qlg_Path_Name_T structure that holds the absolute path name of the current directory. The path name is not null-terminated within the structure. For more information on the Qlg_Path_Name_T structure, see Path name format.

size (Input) The number of bytes allocated for *buf*.

Related Information

- “getcwd()—Get Current Directory” on page 113—Get Current Directory
- “QlgChdir()—Change Current Directory (using NLS-enabled path name)” on page 239—Change Current Directory (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines the current directory:

```
#include <unistd.h>
#include <stdio.h>

main()
{

 #define mypath_cd "/tmp"

 const char US_const[3]= "US";
 const char Language_const[4]="ENU";
 typedef struct pnstruct
```

```
    {
     Qlg_Path_Name_T qlg_struct;
     char pn[1024];  /* This size must be large enough */
                              /* to contain the path name.     */
    };

 struct pnstruct path_cd;
 struct pnstruct path_cwd;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)path name_cd, 0x00, sizeof(struct pnstruct));
  path_cd.qlg_struct.CCSID = 37;
  memcpy(path_cd.qlg_struct.Country_ID,US_const,2);
  memcpy(path_cd.qlg_struct.Language_ID,Language_const,3);
  path_cd.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_cd.qlg_struct.Path_Length = sizeof(mypath_cd)-1;
  path_cd.qlg_struct.Path_Name_Delimiter[0] = '/'
  memcpy(path_cd.pn,mypath_cd,sizeof(mypath_cd)-1);

  if (QlgChdir((Qlg_Path_Name_T *)path name_cd) != 0)
    perror("QlgChdir() error()");
  else
  {
    if (QlgGetcwd(Qlg_Path_Name_T *path_cwd,
                              sizeof(struct pnstruct)) == NULL)
      perror("QlgGetcwd() error");
    else
      printf("Successful change to new current working directory.");
  }
}
```

Output:

```
Successful change to new current working directory.
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgGetPathFromFileID()—Get Path Name of Object from Its File ID (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

Qlg_Path_Name_T *QlgGetPathFromFileID(Qlg_Path_Name_T *buf,
size_t size,Qp0lFID_t fileid);
```

Service Program Name: QP0LLIB2
Default Public Authority: *USE
Threadsafe: Yes

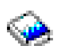
The **QlgGetPathFromFileID()** function, like the **Qp0lGetPathFromFileID()** function, determines an absolute path name of the file identified by *fileid* and stores it in *buf*. The difference is that the **QlgGetPathFromFileID()** function points to a Qlg_Path_Name_T structure, while **Qp0lGetPathFromFileID()** points to a null-terminated character string.

Limited information on the *buf* parameter is provided here. For more information on the *buf* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "Qp0lGetPathFromFileID()—Get Path Name of Object from Its File ID" on page 351—Get Path Name of Object from Its File ID.

Parameters

buf (Output) A pointer to a Qlg_Path_Name_T structure that will be used to hold an absolute path name or a pointer to an absolute path name of the file identified by *fileid*. The path name is not null-terminated within the structure. For more information on the Qlg_Path_Name_T structure, see Path name format.

size (Input) The number of bytes in the buffer `buf`.

fileid (Input) The identifier of the file whose path name is to be returned. This identifier is logged in audit journal entries to identify the file being audited. See the `Parent File ID` and `Object File ID` fields of the audit journal entries described in the iSeries Security Reference  book.

Related Information

- "Qp0lGetPathFromFileID()—Get Path Name of Object from Its File ID" on page 351—Get Path Name of Object from Its File ID

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines the path name of a file, given its file ID. In this example, the *fileid* is hardcoded. More realistically, the *fileid* is obtained from the audit journal entry and passed to **QlgGetPathFromFileID()**.

```
#include <Qp0lstdi.h>
#include <stdio.h>
#include <qtqiconv.h>

void Path_Print(Qlg_Path_Name_T *);

main()
{
  Qp0lFID_t
    fileid = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
              0x00, 0x00, 0x00, 0x00, 0x80, 0xFF, 0xCF, 0x00};

  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[1024]; /* This size must be large enough */
                        /* to contain the path name.      */
  };
  struct pnstruct path;


   /*******************************************************/
   /*   Initialize Qlg_Path_Name_T parameters             */
   /*******************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);

  if (QlgGetPathFromFileID((Qlg_Path_Name_T *)&path,
                 sizeof(struct pnstruct), fileid) == NULL)
```

```
    perror("QlgGetPathFromFileID() error");
  else
  {
    printf("Path retrieved successfully.\n");
    Path_Print((Qlg_Path_Name_T *)&path);
  }
}

void Path_Print(Qlg_Path_Name_T *path_to_print_pointer)
{
 /*************************************************************/
 /* Print a path name that is in the Qlg_Path_Name_T format.   */
 /*************************************************************/

#define PATH_TYPE_POINTER   0x00000001  /* If flag is on,      */
                         /* input structure contains a pointer */
                         /* to the path name, else the path    */
                         /* name is in contiguous storage      */
                         /* within the qlg structure.          */

  typedef union pn_input_type  /* Format of input path name.   */
  {
    char pn_char_type[256]; /* in contiguous storage */
    char *pn_ptr_type;                /* a pointer            */
  };
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    union pn_input_type   pn;
  };
  struct pnstruct *pns;
  char *path_ptr;

  size_t insz;
  size_t outsz = 1000;
  char outbuf[1000];
  char *outbuf_ptr;
  iconv_t cd;
  size_t ret_iconv;

  /* Indicates to convert from ccsid 13488 to 37.              */
  QtqCode_T toCode   =    {37,0,0,0,0,0};
  QtqCode_T fromCode = {13488,0,0,1,0,0};

  if (path_to_print_pointer != NULL)
  {
    /*********************************************************/
    /* Point to and get the size of the path name.           */
    /*********************************************************/
    pns = (struct pnstruct *)path_to_print_pointer;
    if (path_to_print_pointer->Path_Type & PATH_TYPE_POINTER)
      path_ptr = pns->pn.pn_ptr_type;
    else path_ptr = (char *)(pns->pn.pn_char_type);
    insz = pns->qlg_struct.Path_Length;  /* Get path length.*/

    /*********************************************************/
    /* Initialize the print buffer.                          */
    /*********************************************************/
    outbuf_ptr = (char *)outbuf;
    memset(outbuf_ptr, 0x00, insz);

    /*********************************************************/
    /* Use iconv to convert the CCSID.                       */
    /*********************************************************/
    cd = QtqIconvOpen(&toCode,
                     &fromCode);  /* Open a descriptor*/
    if (cd.return_value == -1)
```

```
    { perror("Open conversion descriptor error");
      return;
    }
    if (0 != ((iconv(cd,
                     (char **)&(path_ptr),
                     &insz,
                     (char **)&(outbuf_ptr),
                     &outsz))))
    {
      ret_iconv= iconv_close(cd);/* Close conversion descriptor*/
      perror("Conversion error");
      return;
    }

    /*********************************************************/
    /* Print the name and close the conversion descriptior.    */
    /*********************************************************/
    printf("The file's path is: %s\n",outbuf);
    ret_iconv = iconv_close(cd);
  }                            /* path_to_print_pointer != NULL */
} /* Path_Print */
```

Output:

```
Path retrieved successfully.
The file's path is: /myfile
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgGetpwnam()—Get User Information for User Name (using NLS-enabled path name)

Syntax

```
#include <pwd.h>

struct qplg_passwd *QlgGetpwnam(const char *name);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: No

The **QlgGetpwnam()** function returns a pointer to an object of type `struct qplg_passwd` containing an entry from the user database with a matching *name*.

Parameters

name (Input) User profile name.

The `struct qplg_passwd`, which is defined in the **pwd.h** header file, has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID
uid_t	pw_gid	Group ID
Qlg_Path_Name_T*	pw_dir	Initial working directory
char *	pw_shell	Initial user program

See “getpwnam()—Get User Information for User Name” on page 131 for more on the parameter.

Authorities

*READ authority is required to the user profile associated with the *name*.

Note: Adopted authority is not used.

Return Value

value **QlgGetpwnam** was successful. The return value points to static data that is overwritten on each call to this function. This static storage area is also used by the **QlgGetpwuid()** function.

NULL pointer
QlgGetpwnam was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **QlgGetpwnam()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	The user profile associated with the *name* is currently locked by another process.
[EC2]	Detected pointer that is not valid.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with the *name* was not found.
[ENOMEM (page 543)]	The user profile associated with the *UID* has exceeded its storage limit or is unable to allocate memory.
[EPERM (page 540)]	The calling job does not have *READ authority to the user profile associated with the *name*.
[EUNKNOWN (page 544)]	Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The path name is returned in the default IFS UNICODE CCSID.

Related Information

- The <**pwd.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “getpwnam()—Get User Information for User Name” on page 131—Get User Information for User Name Qlg getpwnam_r
- “getpwnam_r()—Get User Information for User Name” on page 133—Get User Information for User Name
- “QlgGetpwnam_r()—Get User Information for User Name (using NLS-enabled path name)” on page 254—Get User Information for User Name (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the user database information for the user name of MYUSER. The UID is 22. The GID is 1012. The initial working directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <pwd.h>

main()
{
```

```
  struct qplg_passwd *pd;

  if (NULL == (pd = QlgGetpwnam("MYUSER")))
     perror("QlgGetpwnam() error.");
  else
  {
     printf("The user name is: %s\n", pd->pw_name);
     printf("The user id   is: %u\n", pd->pw_uid);
     printf("The group id  is: %u\n", pd->pw_gid);
     printf("The initial working directory length is: %d\n",
             pd->pw_dir->Path_Length);
     printf("The initial working directory CCSID is : %d\n",
             pd->pw_dir->CCSID);
     printf("The initial user program is: %s\n", pd->pw_shell);
  }

}
```

Output:

```
  The user name is: MYUSER
  The user id   is: 22
  The group id  is: 1012
  The initial working directory length is: 24
  The initial working directory CCSID is : 13488
  The initial user program is: *LIBL/QCMD
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgGetpwnam_r()—Get User Information for User Name (using NLS-enabled path name)

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int QlgGetpwnam_r(const char *name,
             struct qplg_passwd *pwd,
                          char *buffer,
                         size_t bufsize,
             struct qplg_passwd **result);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: Yes

The **QlgGetpwnam_r()** function updates the *qplg_passwd* structure pointed to by *pwd* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *name*.

Parameters

name (Input) A pointer to a user profile name.

pwd (Input) A pointer to a qplg_passwd structure.

buffer (Input) A pointer to a buffer from which memory is allocated to hold storage areas referenced by the structure *pwd*.

bufsize
(Input) The size of *buffer* in bytes.

result (Input) A pointer to a location in which a pointer to the updated qplg_passwd structure is stored. If an error occurs or if the requested entry cannot be found, a NULL pointer is stored in this location.

The `struct qplg_passwd`, which is defined in the **pwd.h** header file, has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID
uid_t	pw_gid	Group ID
Qlg_Path_Name_T*	pw_dir	Initial working directory
char *	pw_shell	Initial user program

See “getpwnam_r()—Get User Information for User Name” on page 133 for more on the *pwd*, *result* and other parameters.

Authorities

*READ authority is required to the user profile associated with the *name*.

Return Value

0 **QlgGetpwnam_r** was successful.

Any other value
Failure: The return value contains an error number indicating the error.

Error Conditions

If **QlgGetpwnam_r()** is not successful, the return value usually indicates one of the following errors. Under some conditions, the value could indicate an error other than those listed here.

Error condition	**Additional information**
[EAGAIN (page 541)]	The user profile associated with the *name* is currently locked by another process.
[EC2]	Detected pointer that is not valid.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with the *name* was not found.
[ENOMEM (page 543)]	The user profile associated with the *UID* has exceeded its storage limit or is unable to allocate memory.
[EPERM (page 540)]	The calling job does not have *READ authority to the user profile associated with the *name*.
[ERANGE (page 540)]	Insufficient storage was supplied through *buffer* and *bufsize* to contain the data to be referenced by the resulting group structure.
[EUNKNOWN (page 544)]	Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The path name is returned in the default IFS UNICODE CCSID.

Related Information

- The <**pwd.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “getpwnam()—Get User Information for User Name” on page 131—Get User Information for User Name

- “getpwnam_r()—Get User Information for User Name” on page 133—Get User Information for User Name
- “QlgGetpwnam()—Get User Information for User Name (using NLS-enabled path name)” on page 252—Get User Information for User Name (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the user database information for the user name of MYUSER. The uid is 22. The gid is 1012. The initial working directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <errno.h>

main()
{
  struct qplg_passwd pd;
  qplg_passwd ** tempPwdPtr;
  char pwdbuffer[200];
  int  pwdlinelen = sizeof(pwdbuffer);

  if ((QlgGetpwnam_r("MYUSER",&pd,pwdbuffer,pwdlinelen,tempPwdPtr))!=0   )
     perror("QlgGetpwnam_r() error.");
  else
  {
     printf("\nThe user name is: %s\n", pd->pw_name);
     printf("The user id   is: %u\n", pd->pw_uid);
     printf("The group id  is: %u\n", pd->pw_gid);
     printf("The initial working directory length is: %d\n",
             pd->pw_dir->Path_Length);
     printf("The initial working directory CCSID is : %d\n",
             pd->pw_dir->CCSID);
     printf("The initial user program is: %s\n", pd->pw_shell);
  }

}
```

Output:

```
  The user name is: MYUSER
  The user id   is: 22
  The group id  is: 0
  The intial working directory length is: 24
  The intial working directory CCSID is : 13488
  The initial user program is: *LIBL/QCMD
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgGetpwuid()—Get User Information for User ID (using NLS-enabled path name)

Syntax

```
#include <pwd.h>

struct qplg_passwd *QlgGetpwuid(uid_t uid);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: No

The **QlgGetpwuid()** function returns a pointer to an object of type `struct qplg_passwd` containing an entry from the user database with a matching user ID (*UID*).

Parameters

UID (Input) User ID.

The `struct qplg_passwd`, which is defined in the **pwd.h** header file, has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID
uid_t	pw_gid	Group ID
Qlg_Path_Name_T*	pw_dir	Initial working directory
char *	pw_shell	Initial user program

See "getpwuid()—Get User Information for User ID" on page 135 for more on the parameter.

Authorities

*READ authority is required to the user profile associated with the *UID*.

Note: Adopted authority is not used.

Return Value

value **QlgGetpwuid()** was successful. The return value points to static data that is overwritten on each call to this function. This static storage area is also used by the **QlgGetpwnam()** function.

NULL pointer
QlgGetpwuid() was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **QlgGetpwuid()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	The user profile associated with the *uid* is currently locked by another process.
[EC2]	Detected pointer that is not valid.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with *UID* was not found.
[ENOMEM (page 543)]	The user profile associated with the *UID* has exceeded its storage limit or is unable to allocate memory.
[ENOSPC (page 541)]	Machine storage limit exceeded.
[EPERM (page 540)]	The calling job does not have *READ authority to the user profile associated with the *UID*.
[EUNKNOWN (page 544)]	Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

Th path name is returned in the default IFS UNICODE CCSID

Related Information

- The <**pwd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "getpwuid()—Get User Information for User ID" on page 135—Get User Information for User ID
- "getpwuid_r()—Get User Information for User ID" on page 137—Get User Information for User ID
- "QlgGetpwuid_r()—Get User Information for User ID (using NLS-enabled path name)"—Get User Information for User ID (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the user database information for the uid of 22. The user name is MYUSER. The gid is 1012. The initial working directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <pwd.h>

main()
{
  struct qplg_passwd *pd;

  if (NULL == (pd = QlgGetpwuid(22)))
     perror("QlgGetpwuid() error.");
  else
  {
     printf("The user name is: %s\n", pd->pw_name);
     printf("The user id   is: %u\n", pd->pw_uid);
     printf("The group id  is: %u\n", pd->pw_gid);
     printf("The initial working directory length is: %d\n",
             pd->pw_dir->Path_Length);
     printf("The initial working directory CCSID is : %d\n",
             pd->pw_dir->CCSID);
     printf("The initial user program is: %s\n", pd->pw_shell);
  }

}
```

Output:

```
The user name is: MYUSER
The user id   is: 22
The group id  is: 1012
The intial working directory length is: 24
The intial working directory CCSID is : 13488
The initial user program is: *LIBL/QCMD
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgGetpwuid_r()—Get User Information for User ID (using NLS-enabled path name)

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int QlgGetpwuid_r(uid_t uid,
```

```
struct qplg_passwd *pwd,
             char *buffer,
           size_t bufsize,
struct qplg_passwd **result);
```

Service Program Name: QSYPAPI
Default Public Authority: *USE
Threadsafe: Yes

The **QlgGetpwuid_r()** function updates the *qplg_passwd* structure pointed to by *pwd* and stores a pointer to that structure in the location pointed to by *result*. The structure contains an entry from the user database with a matching *UID*.

Parameters

UID (Input) A pointer to a user ID.

pwd (Input) A pointer to a struct qplg_passwd.

buffer (Input) A pointer to a buffer from which memory is allocated to hold storage areas referenced by the structure qplg_passwd.

bufsize
(Input) The size of *buffer* in bytes.

result (Input) A pointer to a location in which a pointer to the updated qplg_passwd structure is stored. If an error occurs or if the requested entry cannot be found, a NULL pointer is stored in this location.

The `struct qplg_passwd`, which is defined in the **pwd.h** header file, has the following elements:

char *	pw_name	User name
uid_t	pw_uid	User ID
uid_t	pw_gid	Group ID
Qlg_Path_Name_T	pw_dir	Initial working directory
char *	pw_shell	Initial user program

See "getpwuid_r()—Get User Information for User ID" on page 137 for more on the *pwd, result* and other parameters.

Authorities

*READ authority is required to the user profile associated with the *UID*.

Return Value

0 **QlgGetpwuid_r()** was successful.

Any other value
Failure: The return value contains an error number indicating the error.

Error Conditions

If **QlgGetpwuid_r()** is not successful, the error value usually indicates one of the following errors. Under some conditions, the value could indicate an error other than those listed here.

Error condition	**Additional information**
[EAGAIN (page 541)]	The user profile associated with the *uid* is currently locked by another process.

Error condition	Additional information
[EC2]	Detected pointer that is not valid.
[EINVAL (page 540)]	Value is not valid. Check the job log for messages.
[ENOENT (page 540)]	The user profile associated with *UID* was not found.
[ENOMEM (page 543)]	The user profile associated with the *UID* has exceeded its storage limit or is unable to allocate memory.
[ENOSPC (page 541)]	Machine storage limit exceeded.
[EPERM (page 540)]	The calling job does not have *READ authority to the user profile associated with the *UID*.
[ERANGE (page 540)]	Insufficient storage was supplied through *buffer* and *bufsize* to contain the data to be referenced by the resulting group structure.
[EUNKNOWN (page 544)]	Unknown system state. Check the job log for a CPF9872 message. If there is no message, verify that the home directory field in the user profile can be displayed.

Usage Notes

The path name is returned in the default IFS UNICODE CCSID.

Related Information

- The <**pwd.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “getpwuid()—Get User Information for User ID” on page 135—Get User Information for User ID
- “getpwuid_r()—Get User Information for User ID” on page 137—Get User Information for User ID
- “QlgGetpwuid()—Get User Information for User ID (using NLS-enabled path name)” on page 256—Get User Information for User ID (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets the user database information for the uid of 22. The user name is MYUSER. The GID is 1012. The intial working directory is /home/MYUSER. The initial user program is *LIBL/QCMD.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <errno.h>

main()
{
  struct qplg_passwd pd;
  passwd ** tempPwdPtr;
  char pwdbuffer[200];
  int  pwdlinelen = sizeof(pwdbuffer);

  if ((QlgGetpwuid_r(22,&pd,pwdbuffer,pwdlinelen,tempPwdPtr))!=0)
     perror("QlgGetpwuid_r() error.");
  else
  {
     printf("\nThe user name is: %s\n", pd->pw_name);
     printf("The user id   is: %u\n", pd->pw_uid);
     printf("The group id  is: %u\n", pd->pw_gid);
     printf("The initial working directory length is: %d\n",
             pd->pw_dir->Path_Length);
     printf("The initial working directory CCSID is : %d\n",
             pd->pw_dir->CCSID);
     printf("The initial user program is: %s\n", pd->pw_shell);
  }

}
```

Output:

```
The user name is: MYUSER
The user ID   is: 22
The group ID  is: 0
The initial working directory length is: 24
The initial working directory CCSID is : 13488
The initial user program is: *LIBL/QCMD
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgLchown()—Change Owner and Group of Symbolic Link (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgLchown(Qlg_Path_Name_T *path, uid_t owner,gid_t
group);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "lchown()—Change Owner and Group of Symbolic Link" on page 149.

The **QlgLchown()** function, like the **lchown()** function, changes the owner and group of a file. The difference is that the **QlgLchown()** function takes a pointer to a Qlg_Path_Name_T structure, while **lchown()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "lchown()—Change Owner and Group of Symbolic Link" on page 149—Change Owner and Group of Symbolic Link.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file whose owner and group are being changed. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "lchown()—Change Owner and Group of Symbolic Link" on page 149—Change Owner and Group of Symbolic Link
- "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240—Change File Authorizations
- "QlgLstat()—Get File or Link Information (using NLS-enabled path name)" on page 265—Get File or Link Information
- "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293—Get File Information

Example

See Code disclaimer information for information pertaining to code examples.

The following example changes the owner and group of a file:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <Qp0lstdi.h>

main() {

#define mypath_link_name "temp.link"
#define mypath_fn "temp.file"

  const char US_const]3[= "US";
  const char Language_const]4[="ENU";

  struct stat info;
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn]100[; /* This array size must be >= the  */
                         /* length of the path name or this must  */
                         /* be a pointer to the path name.        */
  };
  struct pnstruct path_link;
  struct pnstruct path_fn;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path_link, 0x00, sizeof(struct pnstruct));
  path_link.qlg_struct.CCSID = 37;
  memcpy(path_link.qlg_struct.Country_ID,US_const,2);
  memcpy(path_link.qlg_struct.Language_ID,Language_const,3);
  path_link.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_link.qlg_struct.Path_Length = sizeof(mypath_link_name)-1;
  path_link.qlg_struct.Path_Name_Delimiter]0[ = '/';
  memcpy(path_link.pn,mypath_link_name,sizeof(mypath_link_name)-1);

  memset((void*)&path_fn, 0x00, sizeof(struct pnstruct));
  path_fn.qlg_struct.CCSID = 37;
  memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
  memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
  path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
  path_fn.qlg_struct.Path_Name_Delimiter]0[ = '/';
  memcpy(path_fn.pn,mypath_fn,sizeof(mypath_fn)-1);

  if (QlgSymlink((Qlg_Path_Name_T *)&path_fn,
                (Qlg_Path_Name_T *)&path_link) == -1)
    perror("QlgSymlink() error");
  else {
    QlgLstat((Qlg_Path_Name_T *)&path_link, &info);
    printf("original owner was %d and group was %d\n", info.st_uid,
           info.st_gid);
    if (QlgLchown((Qlg_Path_Name_T *)&path_link, 152, 0) != 0)
      perror("QlgLchown() error");
    else {
      QlgLstat((Qlg_Path_Name_T *)&path_link, &info);
      printf("after QlgLchown(), owner is %d and group is %d\n",
             info.st_uid, info.st_gid);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path_link);
  }
}
```

Output:

```
original owner was 137 and group was 0
after QlgLchown(), owner is 152 and group is 0
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgLink()—Create Link to File (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgLink(Qlg_Path_Name_T *existing, Qlg_Path_Name_T *new);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "link()—Create Link to File" on page 153.

The **QlgLink()** function, like the **link()** function, provides an alternative path name for the existing file so that the file can be accessed by either the existing name or the new name. The difference is that the **QlgLink()** function supports pointers to Qlg_Path_Name_T structures, while **link()** supports pointers to character strings.

Limited information on the *existing* and the *new* parameters is provided here. For more information on these parameters and for a discussion of the authorities required, return values, and related information, see "link()—Create Link to File" on page 153—Create Link to File.

Parameters

existing
(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of an existing file to which a new link is to be created. For more information on the Qlg_Path_Name_T structure, see Path name format.

new
(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name that is the name of the new link. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "link()—Create Link to File" on page 153—Create Link to File
- "Qp0lUnlink()—Remove Link to File" on page 418—Remove Link to File (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **QlgLink()**:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <Qp0lstdi.h>

main()
```

```
{
  int file_descriptor;
  struct stat info;
#define mypath_fn "link.example.file"
#define mypath_ln "link.example.link"

  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path_fn;
  struct pnstruct path_ln;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path_fn, 0x00, sizeof(struct pnstruct));
  path_fn.qlg_struct.CCSID = 37;
  memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
  memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
  path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
  path_fn.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_fn.pn,mypath_fn,sizeof(mypath_fn)-1);

  memset((void*)&path_ln, 0x00, sizeof(struct pnstruct));
  path_ln.qlg_struct.CCSID = 37;
  memcpy(path_ln.qlg_struct.Country_ID,US_const,2);
  memcpy(path_ln.qlg_struct.Language_ID,Language_const,3);
  path_ln.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_ln.qlg_struct.Path_Length = sizeof(mypath_ln)-1;
  path_ln.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_ln.pn,mypath_ln,sizeof(mypath_ln)-1);

  if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)&path_fn, S_IWUSR)) < 0)
    perror("QlgCreat() error");
  else {
    close(file_descriptor);
    puts("before QlgLink()");
    QlgStat((Qlg_Path_Name_T *)&path_fn,&info);
    printf("   number of links is %hu\n",info.st_nlink);
    if (QlgLink((Qlg_Path_Name_T *)&path_fn,
                (Qlg_Path_Name_T *)&path_ln) != 0) {
      perror("QlgLink() error");
      QlgUnlink((Qlg_Path_Name_T *)&path_fn);
    }
    else {
      puts("after QlgLink()");
      QlgStat((Qlg_Path_Name_T *)&path_fn,&info);
      printf("   number of links is %hu\n",info.st_nlink);
      QlgUnlink((Qlg_Path_Name_T *)&path_ln);
      puts("after first QlgUnlink()");
      QlgLstat((Qlg_Path_Name_T *)&path_fn,&info);
      printf("   number of links is %hu\n",info.st_nlink);
      QlgUnlink((Qlg_Path_Name_T *)&path_fn);
    }
  }
}
```

Output:

```
before QlgLink()
   number of links is 1
after QlgLink()
   number of links is 2
after first QlgUnlink()
   number of links is 1
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgLstat()—Get File or Link Information (using NLS-enabled path name)

Syntax

```
#include <sys/stat.h>

int QlgLstat(Qlg_Path_Name_T *path,struct stat *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "lstat()—Get File or Link Information" on page 162.

The **QlgLstat()** function, like the **lstat()** function, gets status information about a specified file and places it in the area of memory pointed to by *buf*. The difference is that the **QlgLstat()** function takes a pointer to a Qlg_Path_Name_T structure, while **lstat()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "lstat()—Get File or Link Information" on page 162—Get File or Link Information.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "lstat()—Get File or Link Information" on page 162—Get File or Link Information
- "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240—Change File Authorizations (using NLS-enabled path name)
- "QlgChown()—Change Owner and Group of File (using NLS-enabled path name)" on page 242—Change Owner and Group of File (using NLS-enabled path name)
- "QlgCreat()—Create or Rewrite File (using NLS-enabled path name)" on page 244—Create or Rewrite File (using NLS-enabled path name)
- "QlgLink()—Create Link to File (using NLS-enabled path name)" on page 263—Create Link to File (using NLS-enabled path name)
- "QlgMkdir()—Make Directory (using NLS-enabled path name)" on page 270—Make Directory (using NLS-enabled path name)
- "QlgReadlink()—Read Value of Symbolic Link (using NLS-enabled path name)" on page 284—Read Value of Symbolic Link (using NLS-enabled path name)
- "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293—Get File Information (using NLS-enabled path name)

- “QlgSymlink()—Make Symbolic Link (using NLS-enabled path name)” on page 300—Make Symbolic Link (using NLS-enabled path name)
- “QlgUtime()—Set File Access and Modification Times (using NLS-enabled path name)” on page 303—Set File Access and Modification Times (using NLS-enabled path name)
- “Qp0lUnlink()—Remove Link to File” on page 418—Remove Link to File (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example provides status information for a file:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <Qp0lstdi.h>

main() {

  struct stat info;
  int file_descriptor;
 #define mypath_fn "temp.file"
 #define mypath_ln "temp.link"

  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this must */
                  /* be a pointer to the path name.       */
  };
 struct pnstruct path_fn;
 struct pnstruct path_ln;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path_fn, 0x00, sizeof(struct pnstruct));
  path_fn.qlg_struct.CCSID = 37;
  memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
  memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
  path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
  path_fn.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_fn.pn,mypath_fn,sizeof(mypath_fn)-1);

  memset((void*)&path_ln, 0x00, sizeof(struct pnstruct));
  path_ln.qlg_struct.CCSID = 37;
  memcpy(path_ln.qlg_struct.Country_ID,US_const,2);
  memcpy(path_ln.qlg_struct.Language_ID,Language_const,3);
  path_ln.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_ln.qlg_struct.Path_Length = sizeof(mypath_ln)-1;
  path_ln.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_ln.pn,mypath_ln,sizeof(mypath_ln)-1);

 if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)&path_fn, S_IWUSR)) < 0)
    perror("QlgCreat() error");
  else {
```

```
    close(file_descriptor);
    if (QlgLink((Qlg_Path_Name_T *)&path_fn,
                (Qlg_Path_Name_T *)&path_ln)
                                   !=0
      perror("QlgLink() error");
    else {
      if (QlgLstat((Qlg_Path_Name_T *)&path_ln, &info) != 0)
        perror("QlgLstat() error");
      else {
        puts("QlgLstat() returned:");
        printf("  inode:   %d\n",   (int) info.st_ino);
        printf(" dev id:   %d\n",   (int) info.st_dev);
        printf("   mode:   %08x\n",       info.st_mode);
        printf("  links:   %d\n",         info.st_nlink);
        printf("    uid:   %d\n",   (int) info.st_uid);
        printf("    gid:   %d\n",   (int) info.st_gid);
      }
      QlgUnlink((Qlg_Path_Name_T *)&path_ln);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path_fn);
  }
}
```

Output:

```
QlgLstat() returned:
  inode:   8477
 dev id:   0
   mode:   00008080
  links:   2
    uid:   1782
    gid:   0
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgLstat64()—Get File or Link Information (large file enabled and using NLS-enabled path name)

Syntax

```
#include <sys/stat.h>

int QlgLstat64(Qlg_Path_Name_T *path, struct stat64 *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "lstat()—Get File or Link Information" on page 162.

The **QlgLstat64()** function, like the **lstat64()** function, gets status information about a specified file and places it in the area of memory pointed to by *buf*. The difference is that the **QlgLstat64()** function takes a pointer to a Qlg_Path_Name_T structure, while **lstat64()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "lstat64()—Get File or Link Information (Large File Enabled)" on page 167—Get File or Link Information (Large File Enabled) or "lstat()—Get File or Link Information" on page 162—Get File or Link Information.

Parameters

path
: (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "lstat64()—Get File or Link Information (Large File Enabled)" on page 167—Get File or Link Information (large file enabled and using NLS-enabled path name)
- "lstat()—Get File or Link Information" on page 162—Get File or Link Information (using NLS-enabled path name)
- "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240—Change File Authorizations (using NLS-enabled path name)
- "QlgChown()—Change Owner and Group of File (using NLS-enabled path name)" on page 242—Change Owner and Group of File (using NLS-enabled path name)
- "QlgCreat()—Create or Rewrite File (using NLS-enabled path name)" on page 244—Create or Rewrite File (using NLS-enabled path name)
- "QlgLink()—Create Link to File (using NLS-enabled path name)" on page 263—Create Link to File (using NLS-enabled path name)
- "QlgMkdir()—Make Directory (using NLS-enabled path name)" on page 270—Make Directory (using NLS-enabled path name)
- "QlgReadlink()—Read Value of Symbolic Link (using NLS-enabled path name)" on page 284—Read Value of Symbolic Link (using NLS-enabled path name)
- "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293—Get File Information (using NLS-enabled path name)
- "QlgSymlink()—Make Symbolic Link (using NLS-enabled path name)" on page 300—Make Symbolic Link (using NLS-enabled path name)
- "QlgUtime()—Set File Access and Modification Times (using NLS-enabled path name)" on page 303—Set File Access and Modification Times (using NLS-enabled path name)
- "Qp0lUnlink()—Remove Link to File" on page 418—Remove Link to File (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example provides status information for a file:

```
#define _LARGE_FILE_API
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <Qp0lstdi.h>

main() {
  struct stat64 info;
  int file_descriptor;
#define mypath_fn "temp.file"
#define mypath_ln "temp.link"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
```

```
  char pn[100]; /* This array size must be >= the  */
                /* length of the path name or must */
                /* be a pointer to the path name.  */
};
struct pnstruct path_fn;
struct pnstruct path_ln;

 /***************************************************************/
 /*   Initialize Qlg_Path_Name_T parameters                     */
 /***************************************************************/
memset((void*)&path_fn, 0x00, sizeof(struct pnstruct));
path_fn.qlg_struct.CCSID = 37;
memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
path_fn.qlg_struct.Path_Name_Delimiter[0] = '/';
memcpy(path_fn.pn,mypath_fn,sizeof(mypath_fn)-);

memset((void*)&path_ln, 0x00, sizeof(struct pnstruct));
path_ln.qlg_struct.CCSID = 37;
memcpy(path_ln.qlg_struct.Country_ID,US_const,2);
memcpy(path_ln.qlg_struct.Language_ID,Language_const,3);
path_ln.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
path_ln.qlg_struct.Path_Length = sizeof(mypath_ln)-1;
path_ln.qlg_struct.Path_Name_Delimiter[0] = '/';
memcpy(path_ln.pn,mypath_ln,sizeof(mypath_ln)-);

if ((file_descriptor = QlgCreat64((Qlg_Path_Name_T *)&path_fn, S_IWUSR)) <
  perror("QlgCreat64() error");
else {
  close(file_descriptor);
  if (QlgLink((Qlg_Path_Name_T *)&path_fn,
              (Qlg_Path_Name_T *)&path_ln) != 0)
    perror("QlgLink() error");
  else {
    if (QlgLstat64((Qlg_Path_Name_T *)&path_ln, &info) != 0)
      perror("QlgLstat64() error");
    else {
      puts("QlgLstat64() returned:");
      printf("  inode:   %d\n",   (int) info.st_ino);
      printf(" dev id:   %d\n",   (int) info.st_dev);
      printf("   mode:   %08x\n",       info.st_mode);
      printf("  links:   %d\n",         info.st_nlink);
      printf("    uid:   %d\n",   (int) info.st_uid);
      printf("    gid:   %d\n",   (int) info.st_gid);
      printf("   size:   %lld\n", (long long) info.st_size);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path_ln);
  }
  QlgUnlink((Qlg_Path_Name_T *)&path_fn);
}
}
```

Output:

```
QlgLstat() returned:
  inode:   258
 dev id:   1
   mode:   00008080
  links:   2
    uid:   137
    gid:   500
   size:   18
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgMkdir()—Make Directory (using NLS-enabled path name)

Syntax

```
#include <sys/stat.h>

int QlgMkdir(Qlg_Path_Name_T *path, mode_t mode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "mkdir()—Make Directory" on page 169.

The **QlgMkdir()** function, like the **mkdir()** function, creates a new, empty directory whose name is defined by *path*. The difference is that the **QlgMkdir()** function takes a pointer to a Qlg_Path_Name_T structure, while **mkdir()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "mkdir()—Make Directory" on page 169—Make Directory.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the directory to be created. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "mkdir()—Make Directory" on page 169—Make Directory
- "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240—Change File Authorizations (using NLS-enabled path name)
- "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293—Get File Information (using NLS-enabled path name)
- "QlgPathconf()—Get Configurable Path Name Variables (using NLS-enabled path name)" on page 278—Get Configurable Path Name Variables (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a new directory:

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
main() {

 #define mypath "new_dir"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  const char mypath_DOT_DOT[3] = "..";

  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
```

```
                  /* length of the path name or this must */
                  /* be a pointer to the path name.       */
  };
  struct pnstruct path;
  struct pnstruct path_DOT_DOT;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);


  memset((void*)&path_DOT_DOT, 0x00, sizeof(struct pnstruct));
  path_DOT_DOT.qlg_struct.CCSID = 37;
  memcpy(path_DOT_DOT.qlg_struct.Country_ID,US_const,2);
  memcpy(path_DOT_DOT.qlg_struct.Language_ID,Language_const,3);
  path_DOT_DOT.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_DOT_DOT.qlg_struct.Path_Length = sizeof(mypath_DOT_DOT)-1;
  path_DOT_DOT.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_DOT_DOT.pn,mypath_DOT_DOT,sizeof(mypath_DOT_DOT)-1);

  if (QlgMkdir((Qlg_Path_Name_T *)&path,
               S_IRWXU|S_IRGRP|S_IXGRP) != 0)
    perror("QlgMkdir() error");
  else if (QlgChdir((Qlg_Path_Name_T *)&path) != 0)
    perror("first QlgChdir() error");
  else if (QlgChdir((Qlg_Path_Name_T *)&path_DOT_DOT) != 0)
    perror("second QlgChdir() error");
  else if (QlgRmdir((Qlg_Path_Name_T *)&path) != 0)
    perror("QlgRmdir() error");
  else
    puts("success!");
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgMkfifo()—Make FIFO Special File (using NLS-enabled path name)

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>
#include <Qlg.h>

int QlgMkfifo(const Qlg_Path_Name_T *path,
             mode_t mode);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "mkfifo()—Make FIFO Special File" on page 175.

The **QlgMkfifo()** function, like the **mkfifo()** function, creates a new FIFO special file whose name is defined by *path*. The difference is that the **QlgMkfifo()** function takes a pointer to a Qlg_Path_Name_T structure, while **mkfifo()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "mkfifo()—Make FIFO Special File" on page 175—Make FIFO Special File.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the FIFO to be created. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "mkfifo()—Make FIFO Special File" on page 175—Make FIFO Special File
- "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240—Change File Authorizations (using NLS-enabled path name)
- "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293—Get File Information (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates a new FIFO:

```
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include <Qlg.h>

void main()
{
 typedef struct pnstruct
    {
     Qlg_Path_Name_T qlg_struct;
     char[100] pn;  /* This size must be >= the path */
                          /* name length or a pointer to   */
                          /* the path name.                */
    };
 struct pnstruct path;

 char *mypath = "/newFIFO";

 /***************************************************************/
 /*   Initialize Qlg_Path_Name_T structure.                     */
 /***************************************************************/
 memset((void*)path name, 0x00, sizeof(struct pnstruct));
 path.qlg_struct.CCSID = 37;
 memcpy(path.qlg_struct.Country_ID, "US", 2);
 memcpy(path.qlg_struct.Language_ID, "ENU", 3);
 path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
 path.qlg_struct.Path_Length = strlen(mypath);
 path.qlg_struct.Path_Name_Delimiter = '/';
 memcpy(path.pn, mypath, strlen(mypath));

 if (QlgMkfifo((Qlg_Path_Name_T *)path name,
              S_IRWXU|S_IRWXO) != 0)
   perror("QlgMkfifo() error");
 else
   puts("success!");

 return;
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgOpen()—Open a File (using NLS-enabled path name)

Syntax

```
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>

int QlgOpen(Qlg_Path_Name_T *Path_Name,
int oflag, . . .);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "open()—Open File" on page 195 API.

The **QlgOpen()** function, like the **open()** function, opens a file or creates a new, empty file whose name is defined by *path* and returns a number called a **file descriptor**. The difference is that the **QlgOpen()** function takes a pointer to a Qlg_Path_Name_T structure, while **open()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, usage notes, return values, and related information, see "open()—Open File" on page 195—Open a File.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file to be opened. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "open()—Open File" on page 195—Open a File
- "QlgCreat()—Create or Rewrite File (using NLS-enabled path name)" on page 244—Create or Rewrite File (using NLS-enabled path name)
- "QlgOpen64()—Open File (large file enabled and using NLS-enabled path name)" on page 274—Open File (large file enabled and using NLS-enabled path name)
- "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293—Get File Information (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates and opens an output file for exclusive access. This program was stored in a source file with CCSID 37, so the constant string ″newfile″ will be compiled in CCSID 37. Therefore, the language and country or region specified are United States English, and the CCSID specified is 37.

```
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>

main()
{
  int fildes;
```

```
  const char US_const[3]= "US";
  const char Language_const[4]="ENU";

  struct pnstruct
    {
     Qlg_Path_Name_T  qlg_struct;
     char             pn[7];
    };
  struct pnstruct pns;
  struct pnstruct *pns_ptr = NULL;

  char fn[]="newfile";

  memset((void*)&pns, 0x00, sizeof(struct pnstruct));
  pns.qlg_struct.CCSID = 37;
  memcpy(pns.qlg_struct.Country_ID,US_const,2);
  memcpy(pns.qlg_struct.Language_ID,Language_const,3);;
  pns.qlg_struct.Path_Type = 0;
  pns.qlg_struct.Path_Length = sizeof(fn) - 1;
  pns.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(pns.pn,fn,sizeof(fn)-1);

  pns_ptr = &pns;
  if(fildes = QlgOpen((Qlg_Path_Name_T *)pns_ptr,
           O_WRONLY|O_CREAT|O_EXCL, S_IRWXU)) == -1)
  {
        perror("QlgOpen() error");
  }

}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgOpen64()—Open File (large file enabled and using NLS-enabled path name)

Syntax

```
#include <fcntl.h>

int QlgOpen64(Qlg_Path_Name_T *path, int oflag, . . .);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "open()—Open File" on page 195.

The **QlgOpen64()** function, like the **open64()** and **open()** functions, opens a file and returns a number called a file descriptor. **QlgOpen64()** differs from **open64()** in that the **QlgOpen64()** function takes a pointer to a Qlg_Path_Name_T structure, while **open64()** takes a pointer to a character string. **QlgOpen64()** differs from **open()** in that it automatically opens a file with the O_LARGEFILE flag set.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "open()—Open File" on page 195—Open a File or "open64()—Open File (Large File Enabled)" on page 211—Open File (Large File Enabled).

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file to be opened. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "open()—Open File" on page 195—Open a File
- "open64()—Open File (Large File Enabled)" on page 211—Open File (Large File Enabled)
- "QlgCreat()—Create or Rewrite File (using NLS-enabled path name)" on page 244—Create or Rewrite File (using NLS-enabled path name)
- "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293—Get File Information (using NLS-enabled path name)

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgOpendir()—Open Directory (using NLS-enabled path name)

Syntax

```
#include <sys/types.h>
#include <dirent.h>

DIR *QlgOpendir(Qlg_Path_Name_T *dirname);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "opendir()—Open Directory" on page 212.

The **QlgOpendir()** function, like the **opendir()** function, opens a directory so it can be read. The difference is that the **QlgOpendir()** function takes a pointer to a Qlg_Path_Name_T structure, while the **opendir()** function takes a pointer to a character string. The **QlgOpendir()** function opens a directory so it can be read with the **QlgReaddir()** function.

Names returned on calls to **QlgReaddir()** are returned in the coded character set identifier (CCSID) specified at the time the directory is opened. **QlgOpendir()** allows the CCSID to be specified in the Qlg_Path_Name_T structure. **opendir()** uses the CCSID that is in effect for the current job at the time the **opendir()** function is called. See "opendir()—Open Directory" on page 212—Open Directory for more on the job CCSID.

Limited information on the *dirname* parameter is provided here. For more information on the *dirname* parameter and for a discussion of authorities required, return values, and related information, see "opendir()—Open Directory" on page 212—Open Directory.

Parameters

dirname

(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the directory to be opened. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "opendir()—Open Directory" on page 212—Open Directory

- "QlgReaddir()—Read Directory Entry (using NLS-enabled path name)" on page 280—Read Directory Entry (using NLS-enabled path name)
- QlgSpawn()—Spawn Process (using NLS-enabled path name)
- QlgSpawnp()—Spawn Process with Path (using NLS-enabled fileh name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example opens a directory:

```
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>

void traverse(char *fn, int indent) {
  DIR *dir;
  int count;
  struct stat info;

  typedef struct my_dirent_lg
  {
    struct dirent_lg *entry;
    char          d_lg_name[1];
  };

  struct my_dirent_lg lg_struct;
  struct dirent_lg *entry;

  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[1025]; /* This array size must be >=   */
                           /* the length of the path name or  */
                           /* this must be a pointer to the   */
                           /* path name.                      */
  };

  struct pnstruct path;
  struct pnstruct path_to_stat;
  char *temp_char_path[1025];

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T structure, since the path name */
   /*   was not in the Qlg_Path_Name_T format when this function  */
   /*   was called.                                               */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  path.qlg_struct.Path_Length = strlen(fn);
  memcpy(path.pn,fn,strlen(fn));

  for (count=0; count < indent; count++) printf("  ");
  printf("%s\n", fn);

  if ((dir = QlgOpendir((Qlg_Path_Name_T *)&path)) == NULL)
    perror("QlgOpendir() error");
```

```
  else
  {
    path_to_stat = path;

    while ((entry = QlgReaddir(dir)) != NULL)
    {

      if
        (entry->d_lg_name[0] != '.')
      {
        /* Concat the components of the path name into a  */
        /* Qlg_Path_Name_T structure that is used on the  */
        /* next function that is called. Clear and        */
        /* use a temporary buffer to ensure that only     */
        /* characters returned by QlgReaddir() are        */
        /* included in the concatenated path name         */
        /* structure.                                     */
        strcpy(path_to_stat.pn,path.pn);
        strcat(path_to_stat.pn, "/");
        memset(temp_char_path, 0x00,1025);
        memcpy(temp_char_path,
               entry->d_lg_name,entry->d_lg_qlg.Path_Length);

        strcat(path_to_stat.pn,(char *)&temp_char_path);

        /* Calculate the size of the path, including the  */
        /* length of the path specified on the open, the  */
        /* length of the name returned by QlgReaddir(),   */
        /* and the delimiter.                             */

        path_to_stat.qlg_struct.Path_Length =
          (path.qlg_struct.Path_Length +
           entry->d_lg_qlg.Path_Length + 1);


        /* Call QlgStat() to determine if the path name   */
        /* is a directory.                                */
        if (QlgStat((Qlg_Path_Name_T *)&path_to_stat,
                    &info) != 0)
        {
          fprintf(stderr, "QlgStat() error on %s: %s\n",
                  path_to_stat.pn,
                  strerror(errno));
        }
        else if (S_ISDIR(info.st_mode))
        {
          /* this a directory so loop to open its objects.*/
          traverse(path_to_stat.pn, indent+1);
        }
        else printf(" %s\n",path_to_stat.pn);
      }
    }
    closedir(dir);
  }
}


main() {
  puts("Directory structure:");
  traverse("/etc", 0);
}
```

Output:

```
Directory structure:
/etc
  /etc/samples
    /etc/samples/IBM
  /etc/IBM
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgPathconf()—Get Configurable Path Name Variables (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

long QlgPathconf(Qlg_Path_Name_T *path, int name);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "pathconf()—Get Configurable Path Name Variables" on page 216.

The **QlgPathconf()** function, like the **pathconf()** function, lets an application determine the value of a configuration variable (*name*) associated with a particular file or directory (*path*). The difference is that the **QlgPathconf()** function takes a pointer to a Qlg_Path_Name_T structure, while **pathconf()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "pathconf()—Get Configurable Path Name Variables" on page 216—Get Configurable Path Name Variables.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name for which the value of the configuration variable is requested. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "fpathconf()—Get Configurable Path Name Variables by Descriptor" on page 92—Get Configurable Path Name Variables by Descriptor
- "pathconf()—Get Configurable Path Name Variables" on page 216—Get Configurable Path Name Variables
- "QlgChown()—Change Owner and Group of File (using NLS-enabled path name)" on page 242—Change Owner and Group of File (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines the maximum number of bytes in a file name:

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
```

```
main() {
  long result;
#define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  errno = 0;
  puts("examining NAME_MAX limit for root filesystem");
  if ((result = QlgPathconf((Qlg_Path_Name_T *)&path,
                                    _PC_NAME_MAX)) == -1)
    if (errno == 0)
      puts("There is no limit to NAME_MAX.");
    else perror("QlgPathconf() error");
  else
    printf("NAME_MAX is %ld\n", result);
}
```

Output:

```
examining NAME_MAX limit for root filesystem
NAME_MAX is 255
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgProcessSubtree()—Process a Path Name (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

int QlgProcessSubtree (
 Qlg_Path_Name_T         *Path_Name,
 uint                     Subtree_level,
 Qp0l_Objtypes_List_t    *Objtypes_array_ptr,
 uint                     Local_remote_obj,
 Qp0l_IN_EXclusion_List_t *IN_EXclusion_ptr,
 uint                     Err_recovery_action,
 Qp0l_User_Function_t    *UserFunction_ptr,
 void                    *Function_CtlBlk_ptr, ...);
```

Service Program Name: QP0LLIB2
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for “Qp0lProcessSubtree()—Process a Path Name” on page 356.

For a description of this function and information on its parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see “Qp0lProcessSubtree()—Process a Path Name” on page 356—Process a Path Name.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgReaddir()—Read Directory Entry (using NLS-enabled path name)

Syntax

```
#include <sys/types.h>
#include <dirent.h>

struct dirent_lg *QlgReaddir(DIR *dirp);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: No; see Usage Notes for “readdir()—Read Directory Entry” on page 443.

The **QlgReaddir()** function, like the **readdir()** function, returns a pointer to a structure describing the next directory entry in the directory stream associated with *dirp*. The difference is that the **QlgReaddir()** function takes a pointer to a `dirent_lg` structure, while **readdir()** takes a pointer to a `dirent` structure.

Limited information on the *dirp* parameter is provided here. For more information on the *dirp* parameter and for a discussion of authorities required, return values, and related information, see “readdir()—Read Directory Entry” on page 443—Read Directory Entry.

Parameters

dirp (Input) A pointer to DIR that refers to the open directory stream to be read. This pointer is returned by **QlgOpendir()**.

A `dirent_lg` structure has the following contents:

char	d_reserved1[16]	Reserved.
unsigned int	d_fileno_gen_id	The generation ID associated with the file ID.
ino_t	d_fileno	The file ID of the file. This number uniquely identifies the object within a file system.
unsigned int	d_reclen	The length of the directory entry in bytes.
int	d_reserved3	Reserved.
char	d_reserved4[6]	Reserved.
char	d_reserved5[2]	Reserved.

Qlg_Path_Name_T	d_lg_name	A Qlg_Path_Name_T that gives the name of a file in the directory. The path name is not null-terminated within the structure. The structure also provides National Language Support information, which includes ccsid, country_id, and language_id. This structure has a maximum length of {_QP0L_DIR_NAME_LG} bytes. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- “readdir()—Read Directory Entry” on page 443—Read Directory Entry
- “QlgOpendir()—Open Directory (using NLS-enabled path name)” on page 275—Open Directory (using NLS-enabled path name)
- “QlgPathconf()—Get Configurable Path Name Variables (using NLS-enabled path name)” on page 278—Get Configurable Path Name Variables (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example reads the contents of a root directory:

```
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <stdio.h>

main() {

  typedef struct my_dirent_lg
  {
    struct dirent_lg *entry;
    char          d_lg_name[1];
  };

  struct my_dirent_lg lg_struct;
  struct dirent_lg *entry;
#define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100];     /* This array size must be >= */
                                /* the length of the path name   */
                                /* or this must be a pointer     */
                                /* to the path name.             */
  };

  struct pnstruct path;
  DIR     *dir;

   /*****************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                       */
   /*****************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);
```

```
  if ((dir = QlgOpendir((Qlg_Path_Name_T *)&path)) == NULL)
    perror("QlgOpendir() error");
  else {
    puts("contents of root:");
    while ((entry = QlgReaddir(dir)) != NULL)
      printf("  %s\n", entry->d_lg_name);
    closedir(dir);
  }
}
```

Output:

```
contents of root:
  .
  ..
  QSYS.LIB
  QDLS
  QOpenSys
  QOPT
  home
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgReaddir_r()—Read Directory Entry (using NLS-enabled path name)

Syntax

```
#include <sys/types.h>
#include <dirent.h>

int QlgReaddir_r(DIR *dirp, struct dirent_lg *entry,
                 struct dirent_lg **result);
```

Service Program Name: QP0LLIBTS
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "readdir_r()—Read Directory Entry" on page 447.

The **QlgReaddir_r()** function, like the **readdir_r()** function, initializes a structure that is referenced by *entry* to represent the next directory entry in the directory stream that is associated with *dirp*. The difference is that the **QlgReaddir_r()** *dirp* parameter points to a `dirent_lg` structure, while the **readdir_r()** *dirp* parameter points to a `dirent` structure.

The QlgReaddir_r functions stores a pointer to the *entry* structure at the location referenced by *result*.

Limited information on the *dirp* parameter, the *entry* parameter, and the *result* parameter is provided here. For more information on these parameters and for a discussion of authorities required, return values, and related information, see "readdir_r()—Read Directory Entry" on page 447—Read Directory Entry.

Parameters

dirp (Input) A pointer to a DIR that refers to the open directory stream to be read. This pointer is returned by **QlgOpendir()**.

entry (Output) A pointer to a `dirent_lg` structure in which the directory entry is to be placed.

result (Output) A pointer to a pointer to a `dirent_lg` structure. Upon successfully reading a directory entry, this `dirent_lg` pointer is set to the same value as *entry*. Upon reaching the end of the directory stream, this pointer is set to NULL.

A dirent_lg structure has the following contents:

char	d_reserved1[16]	Reserved.
unsigned int	d_fileno_gen_id	The generation ID associated with the file ID.
ino_t	d_fileno	The file ID of the file. This number uniquely identifies the object within a file system.
unsigned int	d_reclen	The length of the directory entry in bytes.
int	d_reserved3	Reserved.
char	d_reserved4[6]	Reserved.
char	d_reserved5[2]	Reserved.
Qlg_Path_Name_T	d_lg_name	A Qlg_Path_Name_T structure that gives the name of a file in the directory. The path name is not null-terminated within the structure. The structure also provides National Language Support information, which includes ccsid, country_id, and language_id. This structure has a maximum length of {_QP0L_DIR_NAME_LG} bytes. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "readdir()—Read Directory Entry" on page 443—Read Directory Entry
- "QlgOpendir()—Open Directory (using NLS-enabled path name)" on page 275—Open Directory (using NLS-enabled path name)
- "QlgPathconf()—Get Configurable Path Name Variables (using NLS-enabled path name)" on page 278—Get Configurable Path Name Variables (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example reads the contents of a root directory:

```
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <stdio.h>

main() {
  int return_code;
  DIR *dir;
  struct dirent_lg entry;
  struct dirent_lg *result;

  typedef struct my_dirent_lg
  {
    struct dirent_lg *entry;
    char          d_lg_name[1];
  };
  struct my_dirent_lg lg_struct;

#define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100];  /* This array size must be >=   */
                         /* the length of the path name or this */
                         /* must be a pointer to the path name. */
```

```
  };
  struct pnstruct path;

   /*************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                   */
   /*************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if ((dir = QlgOpendir((Qlg_Path_Name_T *)&path)) == NULL)
    perror("QlgOpendir() error");
  else {
    puts("contents of root:");
    for (return_code = QlgReaddir_r(dir, &entry, &result);
         result != NULL && return_code == 0;
         return_code = QlgReaddir_r(dir, &entry, &result))
      printf("  %s\n", entry.d_lg_name);
    if (return_code != 0)
      perror("QlgReaddir_r() error");
    closedir(dir);
  }
}
```

Output:

```
contents of root:
  .
  ..
  QSYS.LIB
  QDLS
  QOpenSys
  QOPT
  home
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgReadlink()—Read Value of Symbolic Link (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgReadlink(Qlg_Path_Name_T *path, Qlg_Path_Name_T *buf,
size_t bufsiz);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "readlink()—Read Value of Symbolic Link" on page 452.

The **QlgReadlink()** function, like the **readlink()** function, places the contents of the symbolic link *path* in the buffer *buf*. The difference is that the **QlgReadlink()** function uses pointers to Qlg_Path_Name_T structures, while **readlink()** uses pointers to character strings.

Limited information on the *path* parameter, the *buf* parameter, and the *size* parameter is provided here. For more information on these parameters and for a discussion on authorities required, return values, and related information, see "readlink()—Read Value of Symbolic Link" on page 452—Read Value of Symbolic Link.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the symbolic link. For more information on the Qlg_Path_Name_T structure, see Path name format.

buf (Output) A pointer to the area in which the contents of the link should be stored. For more information on the Qlg_Path_Name_T structure, see Path name format.

bufsiz (Input) The size of *buf* in bytes.

Related Information

- "readlink()—Read Value of Symbolic Link" on page 452—Read Value of Symbolic Link
- "QlgLstat()—Get File or Link Information (using NLS-enabled path name)" on page 265—Get File or Link Information (using NLS-enabled path name)
- "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293—Get File Information (using NLS-enabled path name)
- "QlgSymlink()—Make Symbolic Link (using NLS-enabled path name)" on page 300—Make Symbolic Link (using NLS-enabled path name)
- "Qp0lUnlink()—Remove Link to File" on page 418—Remove Link to File

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **QlgReadlink()**:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>

main() {
  int file_descriptor;

 #define mypath_fn "readlink.file"
 #define mypath_sl "readlink.symlink"

 const char US_const[3]= "US";
 const char Language_const[4]="ENU";
 typedef struct pnstruct
    {
     Qlg_Path_Name_T qlg_struct;
     char pn[100]; /* This array size must be >= the length */
                         /* of the path name or this must be a    */
                         /* pointer to the path name.             */

    };

 struct pnstruct path_fn;
 struct pnstruct path_sl;
 struct pnstruct path_buf;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
```

```
  /*************************************************************/
  memset((void*)path name_fn, 0x00, sizeof(struct pnstruct));
  path_fn.qlg_struct.CCSID = 37;
  memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
  memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
  path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
  path_fn.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_fn.pn,mypath_fn,sizeof(mypath_fn)-1);

  memset((void*)path name_sl, 0x00, sizeof(struct pnstruct));
  path_sl.qlg_struct.CCSID = 37;
  memcpy(path_sl.qlg_struct.Country_ID,US_const,2);
  memcpy(path_sl.qlg_struct.Language_ID,Language_const,3);
  path_sl.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_sl.qlg_struct.Path_Length = sizeof(mypath_sl)-1;
  path_sl.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_sl.pn,mypath_sl,sizeof(mypath_sl)-1);

  if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)path name_fn, S_IWUSR)) < 0)
    perror("QlgCreat() error");
  else {
    close(file_descriptor);
    if (QlgSymlink((Qlg_Path_Name_T *)path name_fn,
                   (Qlg_Path_Name_T *)path name_sl) != 0)
      perror("QlgSymlink() error");
    else {
      if (QlgReadlink((Qlg_Path_Name_T *)path name_sl,
                      (Qlg_Path_Name_T *)path name_buf,
                      sizeof(path_buf)) < 0)
        perror("QlgReadlink() error");
      else printf("QlgReadlink() returned '%s' for '%s'\n",
                                 path name_buf.pn,
                                 path name_sl.pn);

      QlgUnlink((Qlg_Path_Name_T *)path name_sl);
    }
    QlgUnlink((Qlg_Path_Name_T *)path name_fn);
  }
}
```

Output:

```
QlgReadlink() returned 'readlink.file' for 'readlink.symlink'
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

int QlgRenameKeep(Qlg_Path_Name_T *old, Qlg_Path_Name_T *new);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for “Qp0lRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists” on page 373.

The **QlgRenameKeep()** function, like the **Qp0lRenameKeep()** function, renames a file or a directory specified by *old* to the name given by *new*. The difference is that the **QlgRenameKeep()** function takes pointers to Qlg_Path_Name_T structures, while **Qp0lRenameKeep()** takes pointers to character strings.

Limited information on the *old* and *new* parameters is provided here. For more information on these parameters and for a discussion of the authorities required, return values, and related information, see “Qp0lRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists” on page 373—Rename File or Directory, Keep ″new″ If It Exists.

Parameters

old
: (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to the path name of the file to be renamed. For more information on the Qlg_Path_Name_T structure, see Path name format.

new
: (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to the path name of the new name for the file. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- “Qp0lRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists” on page 373—Rename File or Directory, Keep ″new″ If It Exists
- “QlgPathconf()—Get Configurable Path Name Variables (using NLS-enabled path name)” on page 278—Get Configurable Path Name Variables (using NLS-enabled path name)
- “QlgRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists (using NLS-enabled path name)” on page 288—Rename File or Directory, Unlink ″new″ If It Exists (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

When you pass two file names to this example, it changes the first file name to the second file name using **QlgRenameKeep()**.

```
#include <Qp0lstdi.h>
#include <stdio.h>

int main(int argc, char **argv)
{

  if ( argc != 3 )
  {
    printf( "Usage: %s old_fn new_fn\n", argv[0]);
    perror ( "Could not rename file" );
  }

  else
  {
    const char US_const[3]= "US";
    const char Language_const[4]="ENU";
    typedef struct pnstruct
    {
      Qlg_Path_Name_T qlg_struct;
                                /*** EXTRA STORAGE MAY BE NEEDED ***/
      char pn[1025];  /* This size must be >= the path */
                                /* name length or a pointer to   */
                                /* the path name.                */
    };
    struct pnstruct path_old;
    struct pnstruct path_new;
```

```
    struct pnstruct *path_old_ptr;
    struct pnstruct *path_new_ptr;


    memset((void*)&path_old, 0x00, sizeof(struct pnstruct));
    path_old_ptr = &path_old;

    path_old.qlg_struct.CCSID = 37;
    memcpy(path_old.qlg_struct.Country_ID,US_const,2);
    memcpy(path_old.qlg_struct.Language_ID,Language_const,3);;
    path_old.qlg_struct.Path_Type = 0;
    path_old.qlg_struct.Path_Length = strlen(argv[1]);
    path_old.qlg_struct.Path_Name_Delimiter[0] = '/';
    memcpy(path_old.pn,argv[1],sizeof(argv[1])-1);


    memset((void*)&path_new, 0x00, sizeof(struct pnstruct));
    path_new_ptr = &path_new;

    path_new.qlg_struct.CCSID = 37;
    memcpy(path_new.qlg_struct.Country_ID,US_const,2);
    memcpy(path_new.qlg_struct.Language_ID,Language_const,3);;
    path_new.qlg_struct.Path_Type = 0;
    path_new.qlg_struct.Path_Length = strlen(argv[2]);
    path_new.qlg_struct.Path_Name_Delimiter[0] = '/';
    memcpy(path_new.pn,argv[2],sizeof(argv[2])-1);


    if (QlgRenameKeep((Qlg_Path_Name_T *)path_old_ptr,
                      (Qlg_Path_Name_T *)path_new_ptr ) != 0)
    {perror ( "Could not rename file." ); }
    else {perror ( "File renamed." ); }
  }

}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

int QlgRenameUnlink(Qlg_Path_Name_T *old, Qlg_Path_Name_T *new);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "Qp0lRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists" on page 379.

The **QlgRenameUnlink()** function, like the **Qp0lRenameUnlink()** function, renames a file or a directory specified by *old* to the name given by *new*. The difference is that the **QlgRenameUnlink()** function takes a pointer to a Qlg_Path_Name_T structure, while **Qp0lRenameUnlink()** takes a pointer to a character string.

Limited information on the *old* and *old* parameters is provided here. For more information on these parameters and for a discussion of the authorities required, return values, and related information, see “Qp0lRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists” on page 379—Rename File or Directory, Unlink ″new″ If It Exists.

Parameters

old (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file to be renamed. For more information on the Qlg_Path_Name_T structure, see Path name format.

new (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the new name of the file. For more information on the Qlg_Path_Name_T structure, Path name format.

Related Information

- “Qp0lRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists” on page 379—Rename File or Directory, Unlink ″new″ If It Exists
- “QlgPathconf()—Get Configurable Path Name Variables (using NLS-enabled path name)” on page 278—Get Configurable Path Name Variables (using NLS-enabled path name)
- “QlgRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists (using NLS-enabled path name)” on page 286—Rename File or Directory, Keep ″new″ If It Exists (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

When you pass two file names to this example, it tries to change the file name from the first name to the second using **QlgRenameUnlink()**.

```
#include <Qp0lstdi.h>
#include <stdio.h>

int main(int argc, char **argv)
{

  if ( argc != 3 )
  {
    printf( "Usage: %s old_fn new_fn\n", argv[0]);
    perror ( "Could not unlink the file" );
  }

  else
  {
    const char US_const[3]= "US";
    const char Language_const[4]="ENU";
    typedef struct pnstruct
    {
      Qlg_Path_Name_T qlg_struct;
                                /*** EXTRA STORAGE MAY BE NEEDED ***/
      char pn[1025];             /* This size must be >= the path */
                                /* name length or a pointer to   */
                                /* the path name.                */
    };
    struct pnstruct path_old;
    struct pnstruct path_new;

    struct pnstruct *path_old_ptr;
    struct pnstruct *path_new_ptr;


    memset((void*)&path_old, 0x00, sizeof(struct pnstruct));
```

```
    path_old_ptr = &path_old;

    path_old.qlg_struct.CCSID = 37;
    memcpy(path_old.qlg_struct.Country_ID,US_const,2);
    memcpy(path_old.qlg_struct.Language_ID,Language_const,3);;
    path_old.qlg_struct.Path_Type = 0;
    path_old.qlg_struct.Path_Length = strlen(argv[1]);
    path_old.qlg_struct.Path_Name_Delimiter[0] = '/';
    memcpy(path_old.pn,argv[1],sizeof(argv[1]));


    memset((void*)&path_new, 0x00, sizeof(struct pnstruct));
    path_new_ptr = &path_new;

    path_new.qlg_struct.CCSID = 37;
    memcpy(path_new.qlg_struct.Country_ID,US_const,2);
    memcpy(path_new.qlg_struct.Language_ID,Language_const,3);;
    path_new.qlg_struct.Path_Type = 0;
    path_new.qlg_struct.Path_Length = strlen(argv[2]);
    path_new.qlg_struct.Path_Name_Delimiter[0] = '/';
    memcpy(path_new.pn,argv[2],sizeof(argv[2]));


    if (QlgRenameUnlink((Qlg_Path_Name_T *)path_old_ptr,
                       (Qlg_Path_Name_T *)path_new_ptr ) != 0)
    {perror ( "Could not unlink the file." ); }
    else {perror ( "File unlinked." ); }
  }

}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgRmdir()—Remove Directory (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgRmdir(Qlg_Path_Name_T *path,);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "rmdir()—Remove Directory" on page 463.

The **QlgRmdir()** function, like the **rmdir()** function, removes a directory, *path*, provided that the directory is empty; that is, the directory contains no entries other than "dot" (**.**) or "dot-dot" (**..**). The difference is that the **QlgRmdir()** function takes a pointer to a Qlg_Path_Name_T structure, while **rmdir()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of authorities required, return values, usage notes, and related information, see "rmdir()—Remove Directory" on page 463—Remove Directory.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the directory to be removed. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "rmdir()—Remove Directory" on page 463—Remove Directory
- "QlgMkdir()—Make Directory (using NLS-enabled path name)" on page 270—Make Directory (using NLS-enabled path name)
- "Qp0lUnlink()—Remove Link to File" on page 418—Remove Link to File (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example removes a directory:

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <Qp0lstdi.h>

main() {

#define mypath_d  "new_dir"
#define mypath_f "new_dir/new_file"

  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path_d;
  struct pnstruct path_f;

  int file_descriptor;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path_d, 0x00, sizeof(struct pnstruct));
  path_d.qlg_struct.CCSID = 37;
  memcpy(path_d.qlg_struct.Country_ID,US_const,2);
  memcpy(path_d.qlg_struct.Language_ID,Language_const,3);
  path_d.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_d.qlg_struct.Path_Length = sizeof(mypath_d)-1;
  path_d.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_d.pn,mypath_d,sizeof(mypath_d)-1);

  memset((void*)&path_f, 0x00, sizeof(struct pnstruct));
  path_f.qlg_struct.CCSID = 37;
  memcpy(path_f.qlg_struct.Country_ID,US_const,2);
  memcpy(path_f.qlg_struct.Language_ID,Language_const,3);
  path_f.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_f.qlg_struct.Path_Length = sizeof(mypath_f)-1;
  path_d.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_f.pn,mypath_f,sizeof(mypath_f)-1);


  if (QlgMkdir((Qlg_Path_Name_T *)&path_d,S_IRWXU|S_IRGRP|S_IXGRP) !
    perror("QlgMkdir() error");
  else if ((file_descriptor = QlgCreat((Qlg_Path_Name_T *)&path_f,S_IWUSR)) <
    perror("QlgCreat() error");
```

```
  else {
    close(file_descriptor);
    QlgUnlink((Qlg_Path_Name_T *)&path_f);
  }

  if (QlgRmdir((Qlg_Path_Name_T *)&path_d) != 0)
    perror("QlgRmdir() error");
  else
    puts("removed!");
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgSaveStgFree()—Save Storage Free (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

int QlgSaveStgFree(
  Qlg_Path_Name_T         *Path_Name,
  Qp0l_StgFree_Function_t *UserFunction_ptr,
  void                    *Function_CtlBlk_ptr);
```

Service Program Name: QP0LLIB3
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "Qp0lSaveStgFree()—Save Storage Free" on page 399.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see "Qp0lSaveStgFree()—Save Storage Free" on page 399—Save Storage Free.

API introduced: V5R1

Top | Backup and Recovery APIs | UNIX-Type APIs | APIs by category

QlgSetAttr()—Set Attributes (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>
int QlgSetAttr
       (Qlg_Path_Name_T     *Path_Name,
        char                *Buffer_ptr,
        uint                 Buffer_Size,
        uint                 Follow_Symlnk, ...);
```

Service Program Name: QP0LLIB3
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "Qp0lSetAttr()—Set Attributes" on page 403.

For a description of this function and information on its parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see "Qp0lSetAttr()—Set Attributes" on page 403—Set Attributes.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgStat()—Get File Information (using NLS-enabled path name)

Syntax

```
#include <sys/stat.h>

int QlgStat(Qlg_Path_Name_T *path,struct stat *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "stat()—Get File Information" on page 468.

The **QlgStat()** function, like the **stat()** function, gets status information about a specified file and places it in the area of memory pointed to by the *buf* argument. The difference is that the **QlgStat()** function takes a pointer to a Qlg_Path_Name_T structure, while **stat()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "stat()—Get File Information" on page 468—Get File Information.

Parameters

path
(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file from which information is required. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "stat()—Get File Information" on page 468—Get File Information
- "QlgStat64()—Get File Information (large file enabled and using NLS-enabled path name)" on page 295—Get File Information (large file enabled and using NLS-enabled path name)
- "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240—Change File Authorizations (using NLS-enabled path name)
- "QlgChown()—Change Owner and Group of File (using NLS-enabled path name)" on page 242—Change Owner and Group of File (using NLS-enabled path name)
- "QlgCreat()—Create or Rewrite File (using NLS-enabled path name)" on page 244—Create or Rewrite File (using NLS-enabled path name)
- "QlgLink()—Create Link to File (using NLS-enabled path name)" on page 263—Create Link to File (using NLS-enabled path name)
- "QlgLstat()—Get File or Link Information (using NLS-enabled path name)" on page 265—Get File or Link Information (using NLS-enabled path name)
- "QlgMkdir()—Make Directory (using NLS-enabled path name)" on page 270—Make Directory (using NLS-enabled path name)
- "QlgReadlink()—Read Value of Symbolic Link (using NLS-enabled path name)" on page 284—Read Value of Symbolic Link (using NLS-enabled path name)
- "QlgSymlink()—Make Symbolic Link (using NLS-enabled path name)" on page 300—Make Symbolic Link (using NLS-enabled path name)
- "QlgUtime()—Set File Access and Modification Times (using NLS-enabled path name)" on page 303—Set File Access and Modification Times (using NLS-enabled path name)
- "Qp0lUnlink()—Remove Link to File" on page 418—Remove Link to File

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets status information about a file:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>

main() {
  struct stat info;
 #define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if (QlgStat((Qlg_Path_Name_T *)&path, &info) != 0)
    perror("QlgStat() error");
  else {
    puts("QlgStat() returned the following information about root f/s:")
    printf("  inode:   %d\n",   (int) info.st_ino);
    printf(" dev id:   %d\n",   (int) info.st_dev);
    printf("   mode:   %08x\n",       info.st_mode);
    printf("  links:   %d\n",         info.st_nlink);
    printf("    uid:   %d\n",   (int) info.st_uid);
    printf("    gid:   %d\n",   (int) info.st_gid);
  }
}
```

Output: note that the following information will vary from system to system.

```
QlgStat() returned the following information about root f/s:
  inode:   0
 dev id:   1
   mode:   010001ed
  links:   3
    uid:   137
    gid:   500
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgStat64()—Get File Information (large file enabled and using NLS-enabled path name)

Syntax

```
#include <sys/stat.h>

int QlgStat64(Qlg_Path_Name_T *path, struct stat64 *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for “stat64()—Get File Information (Large File Enabled)” on page 475.

The**QlgStat64()** function, like the **stat64()** function, gets status information about a specified file and places it in the area of memory pointed to by the *buf* argument. The difference is that the **QlgStat64()** function takes a pointer to a Qlg_Path_Name_T structure, while **stat64()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see “stat64()—Get File Information (Large File Enabled)” on page 475—Get File Information (Large File Enabled).

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file from which information is required. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- “stat()—Get File Information” on page 468—Get File Information
- “stat64()—Get File Information (Large File Enabled)” on page 475—Get File Information (Large File Enabled)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets status information about a file:

```
#define _LARGE_FILE_API

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>

main() {
  struct stat64 info;
 #define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or this must */
                  /* be a pointer to the path name.       */
  };
```

```
  struct pnstruct path;

   /*************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                   */
   /*************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath));

  if (QlgStat64((Qlg_Path_Name_T *)&path, &info) != 0)
    perror("QlgStat64() error");
  else {
    puts("QlgStat64() returned the following information about root f/s:");
    printf("  inode:   %d\n",   (int) info.st_ino);
    printf(" dev id:   %d\n",   (int) info.st_dev);
    printf("   mode:   %08x\n",       info.st_mode);
    printf("  links:   %d\n",         info.st_nlink);
    printf("    uid:   %d\n",   (int) info.st_uid);
    printf("    gid:   %d\n",   (int) info.st_gid);
  }
}
```

Output: note that the following information will vary from system to system.

```
QlgStat64() returned the following information about root f/s:
  inode:   0
 dev id:   1
   mode:   010001ed
  links:   3
    uid:   137
    gid:   500
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgStatvfs()—Get File System Information (using NLS-enabled path name)

Syntax

```
#include <sys/statvfs.h>

int QlgStatvfs(Qlg_Path_Name_T *path, struct statvfs *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "statvfs()—Get File System Information" on page 478.

The **QlgStatvfs()** function, like the **statvfs()** function, gets status information about the file system that contains the file named by the *path* argument. The difference is that the **QlgStatvfs()** function takes a pointer to a Qlg_Path_Name_T structure, while **statvfs()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "statvfs()—Get File System Information" on page 478—Get File System Information.

Parameters

path (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file from which file system information is required. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "statvfs()—Get File System Information" on page 478—Get File System Information
- "QlgStatvfs64()—Get File System Information (64-Bit enabled and using NLS-enabled path name)" on page 298—Get File System Information (64-Bit Enabled and using NLS-enabled path name)
- "QlgChmod()—Change File Authorizations (using NLS-enabled path name)" on page 240—Change File Authorizations (using NLS-enabled path name)
- "QlgChown()—Change Owner and Group of File (using NLS-enabled path name)" on page 242—Change Owner and Group of File (using NLS-enabled path name)
- "QlgCreat()—Create or Rewrite File (using NLS-enabled path name)" on page 244—Create or Rewrite File (using NLS-enabled path name)
- "QlgLink()—Create Link to File (using NLS-enabled path name)" on page 263—Create Link to File (using NLS-enabled path name)
- "QlgUtime()—Set File Access and Modification Times (using NLS-enabled path name)" on page 303—Set File Access and Modification Times (using NLS-enabled path name)
- "Qp0lUnlink()—Remove Link to File" on page 418—Remove Link to File

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets status information about a file system:

```
#include <sys/statvfs.h>
#include <stdio.h>
#include <sys/types.h>

main() {

  struct statvfs info;
 #define mypath "/"
  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

   /*****************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                       */
   /*****************************************************************/
  memset((void*)path name, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if (-1 == QlgStatvfs((Qlg_Path_Name_T *)path name, &info))
```

```
    perror("QlgStatvfs() error");
  else {
    puts("QlgStatvfs() returned the following information");
    puts("about the Root ('/') file system:");
    printf("  f_bsize    : %u\n", info.f_bsize);
    printf("  f_blocks   : %08X%08X\n",
                          *((int *)&info.f_blocks[0]),
                          *((int *)&info.f_blocks[4]));
    printf("  f_bfree    : %08X%08X\n",
                          *((int *)&info.f_bfree[0]),
                          *((int *)&info.f_bfree[4]));
    printf("  f_files    : %u\n", info.f_files);
    printf("  f_ffree    : %u\n", info.f_ffree);
    printf("  f_fsid     : %u\n", info.f_fsid);
    printf("  f_flag     : %X\n", info.f_flag);
    printf("  f_namemax  : %u\n", info.f_namemax);
    printf("  f_pathmax  : %u\n", info.f_pathmax);
    printf("  f_basetype : %s\n", info.f_basetype);
  }
}
```

Output: The following information will vary from file system to file system.

```
QlgStatvfs() returned the following information
about the Root ('/') file system:
  f_bsize    : 4096
  f_blocks   : 00000000002BF800
  f_bfree    : 0000000000091703
  f_files    : 4294967295
  f_ffree    : 4294967295
  f_fsid     : 0
  f_flag     : 1A
  f_namemax  : 255
  f_pathmax  : 4294967295
  f_basetype : "root" (/)
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgStatvfs64()—Get File System Information (64-Bit enabled and using NLS-enabled path name)

Syntax

```
#include <sys/statvfs.h>

int QlgStatvfs64(Qlg_Path_Name_T *path,
               struct statvfs64 *buf
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "statvfs()—Get File System Information" on page 478.

The **QlgStatvfs64()** function, like the **statvfs64()** function, gets status information about the file system that contains the file named by the *path* argument. The difference is that the **QlgStatvfs64()** function takes a pointer to a Qlg_Path_Name_T structure, while **statvfs64()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "statvfs()—Get File System Information" on page 478—Get File System Information.

Parameters

path
: (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file from which file system information is required. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "statvfs()—Get File System Information" on page 478—Get File System Information
- "statvfs64()—Get File System Information (64-Bit Enabled)" on page 483—Get File System Information (64-Bit Enabled)

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets information about a file system.

```
#include <sys/statvfs.h>
#include <stdio.h>
#include <sys/types.h>

main() {

  struct statvfs info;
 #define mypath "/"

  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100];
       /* This array size must be >= the length */
                        /* of the path name or must be a pointer */
                        /* to the path name.                     */
  };
  struct pnstruct path;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';

  memcpy(path.pn,mypath,sizeof(mypath)-1);


  if (-1 == (QlgStatvfs64((Qlg_Path_Name_T *)&path,
                         (struct statvfs64 *)&info)))
  {
    perror("QlgStatvfs64() error");
  }
  else
  {
    puts("QlgStatvfs64() returned the following information");
    puts("about the Root ('/') file system:");
    printf("  f_bsize    : %u\n", info.f_bsize);

    printf("  f_blocks   : %llu\n", info.f_blocks);
```

```
    printf("  f_bfree    : %llu\n", info.f_bfree);

    printf("  f_files    : %u\n", info.f_files);
    printf("  f_ffree    : %u\n", info.f_ffree);
    printf("  f_fsid     : %u\n", info.f_fsid);
    printf("  f_flag     : %X\n", info.f_flag);
    printf("  f_namemax  : %u\n", info.f_namemax);
    printf("  f_pathmax  : %u\n", info.f_pathmax);
    printf("  f_basetype : %s\n", info.f_basetype);
  }
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgSymlink()—Make Symbolic Link (using NLS-enabled path name)

Syntax

```
#include <unistd.h>

int QlgSymlink(
     Qlg_Path_Name_T *pname, Qlg_Path_Name_T *slink);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "symlink()—Make Symbolic Link" on page 485.

The **QlgSymlink()** function, like the **symlink()** function, creates the symbolic link named by *slink* with the value specified by *pname*. The difference is that the **QlgSymlink()** function takes a pointer to a Qlg_Path_Name_T structure, while **symlink()** takes a pointer to a character string.

Limited information on the **pname* and the **slink* parameter is provided here. For more information on these parameters and for a discussion of authorities required, return values, and related information, see "symlink()—Make Symbolic Link" on page 485—Make Symbolic Link.

Parameters

pname (Input) A pointer to a Qlg_Path_Name_T structure that contains a value or a pointer to a value of the symbolic link. For more information on the Qlg_Path_Name_T structure, see Path name format.

slink (Input) A pointer to a Qlg_Path_Name_T structure that contains a name or a pointer to a name of the symbolic link to be created. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "symlink()—Make Symbolic Link" on page 485—Make Symbolic Link
- "QlgLink()—Create Link to File (using NLS-enabled path name)" on page 263—Create Link to File (using NLS-enabled path name)
- "QlgReadlink()—Read Value of Symbolic Link (using NLS-enabled path name)" on page 284—Read Value of Symbolic Link (using NLS-enabled path name)
- "Qp0lUnlink()—Remove Link to File" on page 418—Remove Link to File

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **QlgSymlink()**:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <Qp0lstdi.h>

main() {
  char buf[30];
  int  fd;
 #define mypath_fn "readlink.file"
 #define mypath_sl "readlink.symlink"

  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >=   */
                            /* the length of the path name or  */
                            /* this must be a pointer to the   */
                            /* path name.                      */
  };

  struct pnstruct path_fn;
  struct pnstruct path_sl;
  struct pnstruct path_buf;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path_fn, 0x00, sizeof(struct pnstruct));
  path_fn.qlg_struct.CCSID = 37;
  memcpy(path_fn.qlg_struct.Country_ID,US_const,2);
  memcpy(path_fn.qlg_struct.Language_ID,Language_const,3);
  path_fn.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_fn.qlg_struct.Path_Length = sizeof(mypath_fn)-1;
  path_fn.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_fn.pn,mypath_sl,sizeof(mypath_fn)-1);

  memset((void*)&path_sl, 0x00, sizeof(struct pnstruct));
  path_sl.qlg_struct.CCSID = 37;
  memcpy(path_sl.qlg_struct.Country_ID,US_const,2);
  memcpy(path_sl.qlg_struct.Language_ID,Language_const,3);
  path_sl.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path_sl.qlg_struct.Path_Length = sizeof(mypath_sl)-1;
  path_sl.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path_sl.pn,mypath_sl,sizeof(mypath_sl)-1);

  if ((fd = QlgCreat((Qlg_Path_Name_T *)&path_fn, S_IWUSR))
          < 0)
    perror("QlgCreat() error");
  else {
    close(fd);
    if (QlgSymlink((Qlg_Path_Name_T *)&path_fn,
                   (Qlg_Path_Name_T *)&path_sl) != 0)
      perror("QlgSymlink() error");

    else {
      if (QlgReadlink((Qlg_Path_Name_T *)&path_sl,
```

```
                    (Qlg_Path_Name_T *)&path_buf,
                    sizeof(struct pnstruct))
          < 0)
        perror("QlgReadlink() error");

      else printf("QlgReadlink() returned '%s' for '%s'\n",
                  (Qlg_Path_Name_T *)&path_buf.pn,
                  (Qlg_Path_Name_T *)&path_sl.pn);

      QlgUnlink((Qlg_Path_Name_T *)&path_sl);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path_fn);
  }
}
```

Output:

```
QlgReadlink() returned 'readlink.file' for 'readlink.symlink'
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgUnlink()—Remove Link to File (using NLS-enabled path name)

Syntax

```
#include <Qp0lstdi.h>

int QlgUnlink(Qlg_Path_Name_T *Path_Name);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for "open()—Open File" on page 195.

The **QlgUnlink()** function, similar to the **unlink()** function, removes a directory entry that refers to a file. **QlgUnlink()**differs from **unlink()** in that the *Path_Name* parameter is a pointer to a Qlg_Path_Name_T structure instead of a pointer to a character string.

For more information on the **Path_Name* parameter and a discussion of the authorities required, return values, and related information, see "unlink()—Remove Link to File" on page 492—Remove Link to File.

Parameters

Path_Name
: (Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the object to be unlinked. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "unlink()—Remove Link to File" on page 492—Remove Link to File
- "link()—Create Link to File" on page 153—Create Link to File
- "QlgOpen()—Open a File (using NLS-enabled path name)" on page 273—Open a File (using NLS-enabled path name)
- "QlgRmdir()—Remove Directory (using NLS-enabled path name)" on page 290—Remove Directory (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example removes a link to a file. This program was stored in a source file with CCSID 37, so the constant string ″newfile″ will be compiled in CCSID 37. Therefore, the country or region and language specified are United States English, and the CCSID specified is 37.

```
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>

main() {
  const char US_const[3]= "US";
  const char Language_const[4]="ENU";

  struct pnstruct
    {
     Qlg_Path_Name_T  qlg_struct;
     char             pn[7];
    };
  struct pnstruct pns;
  struct pnstruct *pns_ptr = NULL;

  char fn[]="unlink.file";

  memset((void*)&pns, 0x00, sizeof(struct pnstruct));
  pns.qlg_struct.CCSID = 37;
  memcpy(pns.qlg_struct.Country_ID,US_const,2);
  memcpy(pns.qlg_struct.Language_ID,Language_const,3);;
  pns.qlg_struct.Path_Type = 0;
  pns.qlg_struct.Path_Length = sizeof(fn)-1;
  pns.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(pns.pn,fn,sizeof(fn)-1);

  pns_ptr = &pns;

  if (QlgUnlink((Qlg_Path_Name_T *)&pns) != 0)
  {
      perror("QlgUnlink() error");
  }
  else printf("QlgUnlink() successful");
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QlgUtime()—Set File Access and Modification Times (using NLS-enabled path name)

Syntax

```
#include <utime.h>

int QlgUtime(Qlg_Path_Name_T *path, const struct utimbuf
*times);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for “utime()—Set File Access and Modification Times” on page 497.

The **QlgUtime()** function, like the **utime()** function, sets the access and modification times of *path* to the values in the `utimbuf` structure. The difference is that the **QlgUtime()** function takes a pointer to a Qlg_Path_Name_T structure, while **utime()** takes a pointer to a character string.

Limited information on the *path* parameter is provided here. For more information on the *path* parameter and for a discussion of other parameters, authorities required, return values, and related information, see "utime()—Set File Access and Modification Times" on page 497—Set File Access and Modification Times.

Parameters

path
(Input) A pointer to a Qlg_Path_Name_T structure that contains a path name or a pointer to a path name of the file for which the times should be changed. For more information on the Qlg_Path_Name_T structure, see Path name format.

Related Information

- "utime()—Set File Access and Modification Times" on page 497—Set File Access and Modification Times

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **QlgUtime()**:

```
#include <utime.h>
#include <time.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <Qp0lstdi.h>

main() {
  int file_descriptor;
  struct utimbuf ubuf;
  struct stat info;

 #define mypath "utime.file"

  const char US_const[3]= "US";
  const char Language_const[4] ="ENU";
  typedef struct pnstruct
  {
    Qlg_Path_Name_T qlg_struct;
    char pn[100]; /* This array size must be >= the  */
                  /* length of the path name or must */
                  /* be a pointer to the path name.  */
  };
  struct pnstruct path;

   /***************************************************************/
   /*   Initialize Qlg_Path_Name_T parameters                     */
   /***************************************************************/
  memset((void*)&path, 0x00, sizeof(struct pnstruct));
  path.qlg_struct.CCSID = 37;
  memcpy(path.qlg_struct.Country_ID,US_const,2);
  memcpy(path.qlg_struct.Language_ID,Language_const,3);
  path.qlg_struct.Path_Type = QLG_CHAR_SINGLE;
  path.qlg_struct.Path_Length = sizeof(mypath)-1;
  path.qlg_struct.Path_Name_Delimiter[0] = '/';
  memcpy(path.pn,mypath,sizeof(mypath)-1);

  if ((file_descriptor =
```

```
      QlgCreat((Qlg_Path_Name_T *)&path, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
    puts("before QlgUtime()");
    QlgStat((Qlg_Path_Name_T *)&path,&info);
    printf(" utime.file modification time is %ld\n",
           info.st_mtime);
    ubuf.modtime = 0;       /* set modification time to Epoch */
    time(&ubuf.actime);
    if (QlgUtime((Qlg_Path_Name_T *)&path, &ubuf) != 0)
      perror("QlgUtime() error");
    else {
      puts("after QlgUtime()");
      QlgStat((Qlg_Path_Name_T *)&path,&info);
      printf(" utime.file modification time is %ld\n",
             info.st_mtime);
    }
    QlgUnlink((Qlg_Path_Name_T *)&path);
  }
}
```

Output:

```
before QlgUtime()
  utime.file modification time is 749323571
after QlgUtime()
  utime.file modification time is 0
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

Perform Miscellaneous File System Functions (QP0FPTOS) API

Required Parameter Group:

1	Function type	Input	Char(*)
2	Function extension 1	Input	Char(*)
3	Function extension 2	Input	Char(*)

Default Public Authority: *USE
Threadsafe: No

The Perform Miscellaneous File System Function (QP0FPTOS) API is used to perform a variety of file system functions. The first parameter defines the type of function that is requested. Other parameters are optional, depending on the selected function. The output from this API varies, based on the selected function. See the function descriptions for more details.

Authorities and Locks

To call this program you must have *SERVICE special authority, or be authorized to the Service Dump function of i5/OS through iSeries Navigator's Application Administration support. The Change Function Usage (CHGFCNUSG) command or Change Function Usage Information (QSYCHFUI) API, with a function ID of QIBM_SERVICE_DUMP, also can be used to change the list of users allowed to perform dump operations.

Note: Adopted authority is not used.

Required Parameter Group

Required parameters vary according to the selected function. The selected function is identified by the first parameter on the call to the API.

Function Type

INPUT; CHAR(*)

The desired file system function to perform. Valid values follow:

*(1) *DUMP*

Creates a general file system dump in a spooled file with file name ″QSYSPRT″ and with ″QP0FDUMP″ in the User Data field. No other parameters are required or supported when *DUMP is specified.

*(2) *DUMPALL*

Creates a variety of file system dumps in a single spooled file with file name ″QSYSPRT″ and with ″QP0FDUMP″ in the User Data field. The following table describes the optional parameter when *DUMPALL is specified.

Function	Function extension 1	Function extension 2	Description
*DUMPALL	Job number (CHAR 6)	(Not supported)	Specifies the job that is dumped. If a job is not specified, the data is dumped for all jobs. If there are multiple jobs with the same number, the first one encountered will be dumped.

*(3) *DUMPLFS*

Creates a dump of logical file system data in a spooled file with file name ″QSYSPRT″ and with ″QP0FDUMP″ in the User Data field. The following table describes the optional parameter when *DUMPLFS is specified.

Function	Function extension 1	Function extension 2	Description
*DUMPLFS	Job number (CHAR 6)	(Not supported)	Specifies the job that is dumped. If a job is not specified, the data is dumped for all jobs. If there are multiple jobs with the same number, the first one encountered will be dumped.

*(4) *NFSFORCE*

Sets various values and modes for the network file system. The following table describes the required parameters when *NFSFORCE is specified.

Function	Function extension 1	Function extension 2	Description
*NFSFORCE	V2	ON or OFF	If ON, indicates version 2 mounts only by the client. If QNFSMNTD is started afterwards, then server will permit version 2 mounts only.

*(5) *REBUILDDEVNULL*
: Attempts to create the /dev/null and dev/zero character special files. If an existing dev/null or dev/zero object exists that is not a character special file, then the object is renamed to /dev/null.prv or dev/zero.prv. If /dev/null.prv or /dev/zero.prv exists, then it it renamed to /dev/null.prv.001 or /dev/zero.prv.001, /dev/null.prv.002 or /dev/zero.prv.002, and so on, until a name is found for the object. If 999 is exceeded and the rename cannot be done, the object is not renamed and an informational message is issued and the QP0FPTOS program completes successfully. No other parameters are required or supported when *REBUILDDEVNULL is specified.

*(6) *TRACE6ON or *TRACE6OFF*
: *TRACE6ON starts the logging of trace messages in the user job log for some network file system functions. *TRACE6OFF stops the logging of these messages.

*(7) *TRACE8ON or *TRACE8OFF*
: *TRACE8ON starts the logging of trace messages to the QSYSOPR message queue for some network file system functions. *TRACE8OFF stops the logging of these messages.

*(8) *TRACE9ON or *TRACE9OFF*
: *TRACE9ON starts the collection of some network file system statistics and resets the statistics. *TRACE9OFF stops the collection of these statistics.

*(9) *DUMPNFSSTATS*
: Creates a file system dump of network file system (NFS) statistics (both client and server) in a spooled file with file name ″QSYSPRT″ and with ″QP0FDUMP″ in the User Data field. The information dumped comes from a window of time specified with the *TRACE9ON/OFF function. No other parameters are required or supported when *DUMPNFSSTATS is specified.

Function extension 1
: INPUT; CHAR(*)

Function extension 1 is optional or required, based on the first parameter. Whenever it is valid, function extension 1 is described above along with a first parameter description. Function extension 1 is valid when the first parameter is listed below:

*(1) *DUMPALL*

*(2) *DUMPLFS*

*(3) *NFSFORCE*

Function extension 2
: INPUT; CHAR(*)

Function extension 2 is optional or required, based on the first parameter. Whenever it is valid, function extension 2 is described above along with a first parameter description. Function extension 2 is valid when the first parameter is listed below:

*(1) *NFSFORCE*

Usage Notes

If this API is called without the first parameter that is required, then message CPFBC53 is issued to the caller. This message specifies a parameter that is not valid. To recover, the caller is pointed to the API documentation.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA0A0 E	Object name already exists.
CPFA0D4 E	File system error occurred. Error number &1.
CPDA0FF E	Program not called. You need *SERVICE authority to call this program.
CPFBC53 E	Invalid parameter.
CPFBC54 E	Not authorized to call program.

Examples

See Code disclaimer information for information pertaining to code examples.

CALL QP0FPTOS *DUMP
CALL QP0FPTOS (*DUMPALL '055229')
CALL QP0FPTOS (*DUMPLFS '055229')
CALL QP0FPTOS (*NFSFORCE V2 ON)
CALL QP0FPTOS *REBUILDDEVNULL
CALL QP0FPTOS *TRACE6ON
CALL QP0FPTOS *TRACE6OFF
CALL QP0FPTOS *TRACE8ON
CALL QP0FPTOS *TRACE8OFF
CALL QP0FPTOS *TRACE9ON
CALL QP0FPTOS *TRACE9OFF
CALL QP0FPTOS *DUMPNFSSTATS

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

Qp0lCvtPathToQSYSObjName()— Resolve Integrated File System Path Name into QSYS Object Name

Syntax

```
#include <qp0lstdi.h>

void Qp0lCvtPathToQSYSObjName(
                              Qlg_Path_Name_T *path_name,
                              void            *qsys_info,
                              char             format_name[8],
                              uint             bytes_provided,
                              uint             desired_CCSID,
                              void            *error_code);
```

Service Program Name: QP0LLIB2
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 312.

The **Qp0lCvtPathToQSYSObjName()** function resolves a given integrated file system path name into the four-part QSYS.LIB or independent ASP QSYS.LIB file system name. The primary three parts of the path name are the following components: library, object, and member. The fourth part of the path name is a character representation of the ASP associated with the object, or the independent ASP name. This depends on whether the path refers to an object in the QSYS.LIB file system or an object in an independent ASP QSYS.LIB file system. If the path contains symbolic links, they will be resolved. If, after symbolic links have been resolved, the path does not refer to an object that could be in either the QSYS.LIB file system or an independent ASP QSYS.LIB file system, the API will return with the error message CPFA0DB indicated in the *error_code* structure. Note that the API does not verify that the object exists.

The API also handles wildcard (*) characters in the path name. If the name or type of a library, object, or member is just an asterisk, *ALL is returned as the name or the type. If an asterisk is part of a library, object, or member name, a name containing an asterisk is returned. For example if the following path name is passed in:

/qsys.lib/test*.file/*.*

the API will return:
- Library name: QSYS
- Library type: *LIB
- Object name: TEST*
- Object type: *FILE
- Member name: *ALL
- Member type: *ALL
- ASP name: *SYSBAS

Note that path name components that follow one containing a wildcard character are ignored.

If less than 8 bytes are supplied for the *error_code* structure, errors will cause an exception to be returned to the caller.

Parameters

path_name
: (Input) The path name that refers to the QSYS.LIB or independent ASP QSYS.LIB file system object. The path name must refer to an object on the local file system; this API does not recognize file system objects accessed remotely. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path name format. If the path_name parameter is NULL or points to invalid storage, a CPFA0CE error message is returned.

qsys_info
: (Output) A pointer of type `void *` that refers to a structure that contains the object name. The format of the data returned is specified by the *format_name* parameter. If the qsys_info parameter is NULL or points to invalid storage, a CPF24B4 error message is returned.

format_name
: (Input) An 8-byte character array that indicates how the data will be formatted in the *qsys_info* parameter that is returned. The format is as follows:

  QSYS0100
  : For the format of this structure, see the section “Returned Data Format” on page 310.

If the format_name parameter is NULL or points to invalid storage, a CPF24B4 error message is returned.

bytes_provided
(Input) The number of bytes of data provided in the structure referred to by the *qsys_info* parameter. This value must be at least 8, or a CPF3C24 error message will be returned.

desired_CCSID
(Input) The CCSID the returned object names and types should be converted to. If the value of this parameter is 0, the object names and types will be returned in the job CCSID.

Error code
I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

Authorities

Note: Adopted authority is not used.

Authorization Required for the Qp0lCvtPathToQSYSObjName() API

Object Referred to	Authority Required	Message ID
Each directory, preceding the last component, in the path name.	*X	CPFA09C
Object in the QSYS.LIB or independent ASP QSYS.LIB file system that the path name refers to.	None	None

Returned Data Format

The following table describes the format of the data returned in the *qsys_info* parameter if the QSYS0100 format is specified. For details on the fields of the structure, see the section "Field Descriptions" on page 311.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Bytes_Returned
4	4	BINARY(4)	Bytes_Available
8	8	BINARY(4)	CCSID_Out
12	C	CHAR(28)	Lib_Name
40	28	CHAR(20)	Lib_Type
60	3C	CHAR(28)	Obj_Name
88	58	CHAR(20)	Obj_Type
108	6C	CHAR(28)	Mbr_Name
136	88	CHAR(20)	Mbr_Type
156	9C	CHAR(28)	Asp_Name

Field Descriptions

ASP Name. The path name component that represents the ASP name, if part of the path, or the ASP that the path is associated with. For paths that refer to objects in independent ASP QSYS.LIB file systems, this will be the name of the ASP device description object. For paths that refer to objects in the QSYS.LIB file system, the value of ASP Name will be *SYSBAS.

Bytes_Available. The total number of bytes required to hold all of the data available in the *qsys_info* parameter.

Bytes_Returned. The number of bytes actually returned in the caller's buffer for the *qsys_info* parameter.

CCSID_Out. The CCSID that the returned text is in. This may be different than the *desired_CCSID* if conversion failed. The text is internally normalized, then converted to the desired CCSID. If this conversion from the normalized form does not succeed, the text will be returned in the CCSID of the normalized form.

Lib_Name. The name of the library that the path name refers to. This field is NULL terminated.

Lib_Type. The type of the object, beginning with an * (asterisk). This field will return either *LIB or *ALL. This field is NULL terminated.

Mbr_Name. The name of the member that the path name refers to. This field is NULL terminated, and could be all NULL (all x'00').

Mbr_Type. The type of the member that the path name refers to. This field is NULL terminated. This field will contain *MBR, *ALL, or all NULL (all x'00').

Obj_Name. The name of the object that the path name refers to. This field is NULL terminated, and could be all NULL (all x'00').

Obj_Type. The type of the object that the path name refers to. This field is NULL terminated. This field could contain an object type (for example *FILE), *ALL, or be NULL (all x'00').

The Lib_Name, Lib_Type, Obj_Name, Obj_Type, Mbr_Name, and Mbr_Type fields of the Qp0l_QSYS_Info_t structure will be filled in as appropriate.

If the object that the path name refers to is a library (*LIB), then the lib_name and lib_type fields will contain that library name and *LIB, respectively, and the Obj_Name and Mbr_Name fields will be NULL (all x'00').

If the object name is not an *FILE object with members, then the Mbr_Name field is NULL (all x'00').

If the object name contains quoted strings, the characters within the strings will not be converted to uppercase.

Error Conditions

None.

Error Messages

Message ID	Error Message Text
CPE3101 E	I/O exception non-recoverable error.
CPE3101 E	I/O exception non-recoverable error.
CPE3418 E	Possible APAR condition or hardware failure.
CPE3474 E	Unknown system state.

Message ID	Error Message Text
CPF24B4 E	Severe error while addressing parameter list.
CPF3BF6 E	Path type value not valid.
CPF3C24 E	Length of the receiver variable is not valid.
CPF3CF1 E	Error code parameter not valid.
CPF9872 E	Program &1 in library &2 ended. Reason code is &3.
CPFA092 E	Path name not converted.
CPFA09C E	Not authorized to object. Object is &1.
CPFA09E E	Object in use. Object is &1.
CPFA09F E	Object damaged. Object is &1.
CPFA0A1 E	An input or output error occurred.
CPFA0A2 E	Information passed to this operation was not valid.
CPFA0A3 E	Path name resolution causes looping.
CPFA0A7 E	Path name too long.
CPFA0A8 E	Operation not allowed in a job running multiple threads.
CPFA0A9 E	Object not found. Object is &1.
CPFA0AA E	Error occurred while attempting to obtain space.
CPFA0AD E	Function not supported by file system.
CPFA0B1 E	Requested operation not allowed. Access problem.
CPFA0C0 E	Buffer overflow occurred.
CPFA0C1 E	CCSID &1 not valid.
CPFA0CE E	Error occurred with path name parameter specified.
CPFA0D4 E	File system error occurred. Error number &1.
CPFA0D9 E	Character string not converted.
CPFA0DB E	Object not a QSYS.LIB object. Object is &1.
CPFA0DD E	Function was interrupted.
CPFA0E0 E	File ID conversion of a directory failed.
CPFA0E1 E	The file ID table is damaged.
CPFA0E2 E	System unable to establish a communications connection to a file server.
CPFA0E4 E	The communications connection with the file server was abnormally ended.
CPFA0E5 E	The communications connection with the file server was abnormally ended.
CPFA0E6 E	Object handle rejected by file server.
CPFA0E7 E	System cannot establish a communications connection with a file server.
CPFA1C5 E	Object is a read only object. Object is &1.

Usage Notes

1. This API will fail and return the error message CPFA0A8 when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined file system
     - QSYS.LIB
     - Independent ASP QSYS.LIB
2. This API ignores trailing blank spaces at the end of a path name.

   For example, if the path name is

   ```
   "/qsys.lib/fred.lib/foo.file/abc.mbr      "
   ```

   the trailing blank spaces will be ignored. Thus, the above path name is equivalent to

```
"/qsys.lib/fred.lib/foo.file/abc.mbr"
```

Related Information

- The <**qp0lstdi.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “QlgCvtPathToQSYSObjName()— Resolve Integrated File System Path Name into QSYS Object Name (using NLS-enabled path name)” on page 247— Resolve Integrated File System Path Name into QSYS Object Name

Example

See Code disclaimer information for information pertaining to code examples.

The following example program gets the three-part QSYS name from an integrated file system path name passed to it.

```
#include <qp0lstdi.h>       /* For Qp0lCvtPathToQSYSObjName        */
                            /*     type Qp0l_QSYS_Info_t           */
                            /*     type Qlg_Path_Name_T            */
#include <qusec.h>          /* For type Qus_EC_T                   */
#include <stdlib.h>
#include <stdio.h>

int main ()
{
    /****************************************************************/
    /* Declaration of path_name parameter                           */
    /****************************************************************/
    char             path_info_array[500];
    Qlg_Path_Name_T *path_name;
    const char       fname[] =
            "/qsys.lib/jerold.lib/qcsrc.file/testconv.mbr";
    const char       US_const[] = "US";
    const char       Language_const[] = "ENU";
    const char       Path_Name_Del_const[] = "/";

    /****************************************************************/
    /* Declaration of qsys_info parameter                           */
    /****************************************************************/
    Qp0l_QSYS_Info_t qsys_info;

    /****************************************************************/
    /* Declaration of format_name parameter                         */
    /****************************************************************/
    char format_name[8] = "QSYS0100";

    /****************************************************************/
    /* Declaration of bytes_provided parameter                      */
    /****************************************************************/
    uint bytes_provided;

    /****************************************************************/
    /* Declaration of desired_CCSID parameter.                      */
    /****************************************************************/
    uint desired_CCSID;

    /****************************************************************/
    /* Declarations for error_code parameter                        */
    /****************************************************************/
    Qus_EC_t error_code;
    char     error_string[8];

    /****************************************************************/
    /* Initialize path_name parameter                               */
    /****************************************************************/
    memset(path_info_array, 0, sizeof(path_info_array));
```

```
path_name = (Qlg_Path_Name_T *) path_info_array;

path_name->CCSID = 37;
memcpy(path_name->Country_ID, US_const, 2);
memcpy(path_name->Language_ID, Language_const, 3);
path_name->Path_Type = 0;
path_name->Path_Length = strlen(fname);
memcpy(path_name->Path_Name_Delimiter, Path_Name_Del_const, 1);
memcpy( &(((char *) path_name)[sizeof(Qlg_Path_Name_T)]),
        fname,
        strlen(fname));



/****************************************************************/
/* Initialize qsys_info parameter                               */
/****************************************************************/

/* No initialization requirements for this parameter. */

/****************************************************************/
/* Initialize format_name parameter                             */
/****************************************************************/

/* No additional initialization required. */

/****************************************************************/
/* Initialize bytes_provided parameter.                         */
/****************************************************************/
bytes_provided = sizeof(Qp0l_QSYS_Info_t);

/****************************************************************/
/* Initialize desired_CCSID parameter.                          */
/****************************************************************/
desired_CCSID = 37;

/****************************************************************/
/* Initialize error_code param                                  */
/****************************************************************/
memset(&error_code, 0, sizeof(error_code));
error_code.Bytes_Provided = sizeof(error_code);

/****************************************************************/
/* Call API                                                     */
/****************************************************************/
Qp0lCvtPathToQSYSObjName(path_name,
                        QSYS.LIB_info,
                         format_name,
                         bytes_provided,
                         desired_CCSID,
                        &error_code);

if (error_code.Bytes_Available > 0)
{
    /************************************************************/
    /* Error occurred.                                          */
    /************************************************************/

    printf ("Error occurred: ");
    memcpy (error_string, error_code.Exception_Id, 7);
    error_string[7] = '\0';
    printf ("%s\n", error_string);
    printf ("Bytes available in error code structure: %d.\n",
    error_code.Bytes_Available);
    exit(1);
}
```

```
    /*************************************************************/
    /* API returned successfully.                                */
    /*************************************************************/

    printf ("Library name: %s\n", qsys_info.Lib_Name);
    printf ("Library type: %s\n", qsys_info.Lib_Type);
    printf ("Object name:  %s\n", qsys_info.Obj_Name);
    printf ("Object type:  %s\n", qsys_info.Obj_Type);
    printf ("Member name:  %s\n", qsys_info.Mbr_Name);
    printf ("Member type:  %s\n", qsys_info.Mbr_Type);
    printf ("Asp name:     %s\n", qsys_info.Asp_Name);
    exit(0);
}
```

Output:

```
Library name: JEROLD
Library type: *LIB
Object name:  QCSRC
Object type:  *FILE
Member name:  TESTCONV
Member type:  *MBR
Asp name:     *SYSBAS
```

API introduced: V4R3

Top | UNIX-Type APIs | APIs by category

Perform File System Operation (QP0LFLOP) API

Required Parameter Group:

1	File System Operation	Input	Binary(4), Unsigned
2	Input Buffer	Input	Char(*)
3	Length of input buffer	Input	Binary(4), Unsigned
4	Output Buffer	Output	Char(*)
5	Length of output buffer	Input	Binary(4), Unsigned
6	Error code	I/O	Char(*)

Default Public Authority: *USE
Threadsafe: No

The Perform File System Operation (QP0LFLOP) API performs miscellaneous file system operations.

Authorities and Locks

The authorities required vary for each operation:

(1) QP0L_RETRIEVE_NETGROUP_FILE_ENTRIES

- The user must have execute (*X) data authority to the /etc directory (if it exists).
- The user must have read (*R) data authority to the /etc/netgroup file (if it exists).

(2) QP0L_WRITE_NETGROUP_FILE_ENTRIES

- The user must have write and execute (*WX) data authority to the /etc directory (if it exists).
- The user must have read and write (*RW) data authority to the /etc/netgroup file (if it exists).

(3) QP0L_RETRIEVE_REMOTE_EXPORTS

No special authority required.

(4) QP0L_RETRIEVE_MOUNTED_FILE_SYSTEMS

No special authority required.

Note: Adopted authority is not used.

Required Parameter Group

The following parameters are required.

File system operation
INPUT; BINARY(4), UNSIGNED

The desired file system operation to perform.

You can specify one of the following operations:

(1) QP0L_RETRIEVE_NETGROUP_FILE_ENTRIES
Returns information about all netgroup definitions currently defined in the /etc/netgroup file.

(2) QP0L_WRITE_NETGROUP_FILE_ENTRIES
Recreates the /etc/netgroup file with only the entries provided.

(3) QP0L_RETRIEVE_REMOTE_EXPORTS
Returns all of the Network File System (NFS) exports for a given server.

(4) QP0L_RETRIEVE_MOUNTED_FILE_SYSTEMS
Returns a list of mounted file systems for the local machine along with certain properties of each.

Input buffer
INPUT; CHAR(*)

Information that is required for a given file system operation. The input buffer parameter should be set as follows:

(1) QP0L_RETRIEVE_NETGROUP_FILE_ENTRIES
NULL (no input buffer is required).

(2) QP0L_WRITE_NETGROUP_FILE_ENTRIES
FLOP0200 structure containing the new netgroup entries. For a detailed description of this structure, see "Format of FLOP0200 Structure" on page 320.

(3) QP0L_RETRIEVE_REMOTE_EXPORTS
FLOP0300_INPUT structure containing the remote Network File System (NFS) server name to query the exports from. For a detailed description of this structure, see "Format of FLOP0300 Input Structure" on page 321.

(4) QP0L_RETRIEVE_MOUNTED_FILE_SYSTEMS
FLOP0400_INPUT structure containing the selective filtering information for the mounted file systems requested. For a detailed description of this structure, see "Format of FLOP0400 Input Structure" on page 321.

Length of input buffer
INPUT;BINARY(4), UNSIGNED

The length of the input buffer provided. The length of the input buffer parameter may be specified up to the size of the input buffer area specified by the user program. The length of the input buffer should be 0 when the input buffer is NULL.

Output buffer
OUTPUT; CHAR(*)

Information that is provided by a given file system operation. The output buffer parameter should be set as follows:

(1) QP0L_RETRIEVE_NETGROUP_FILE_ENTRIES

FLOP0100 structure containing enough space to hold all netgroup entries in the /etc/netgroup file. For a detailed description of this structure, see "FLOP0100 Structure Description" on page 318. No partial entries will be returned. To determine if all of the entries were returned, the following semantics will be used:

- If the /etc/netgroup file has no entries defined, bytes available and bytes returned will both be set to 12.
- If the /etc/netgroup file has at least one entry defined, then the bytes available will be greater than 12.
- If all of the defined entries in the /etc/netgroup file could not be returned, then the bytes available will not have the same value as bytes returned.

For example, if the /etc/netgroup file is empty, then bytes available and bytes returned would both be equal to 12. For a different example, if the /etc/netgroup file is not empty, but the length of the output buffer is less than what is required to hold all entries in the /etc/netgroup file, then bytes available would be greater than 12 and bytes returned would be set to 12.

(2) QP0L_WRITE_NETGROUP_FILE_ENTRIES

NULL (no output buffer is required).

(3) QP0L_RETRIEVE_REMOTE_EXPORTS

FLOP0300 structure containing enough space to hold all the export entries from the remote server. For a detailed description of this structure, see "FLOP0300 Output Structure Description" on page 318. No partial entries will be returned. To determine if all of the entries were returned, the following semantics will be used:

- If the server has no exports to return, bytes available and bytes returned will both be set to 12.
- If the server is returning at least one export, then the bytes available will be greater than 12.
- If all of the exports given by the server could not be returned in the space provided, then the bytes available will **not** have the same value as bytes returned. To retrieve all the entries, the request should be made again using an output buffer of at least this size.

(4) QP0L_RETRIEVE_MOUNTED_FILE_SYSTEMS

FLOP0400 structure containing enough space to hold each of the returned mounted file system entries. For a detailed description of this structure, see "FLOP0400 Output Structure Description" on page 319. No partial entries will be returned. To determine if all of the entries were returned, the following semantics will be used:

- If there are no mounted file systems meeting the request criteria, bytes available and bytes returned will both be set to 12.
- If there exists mounted file systems that match the request criteria, then the bytes available will be greater than 12.
- If all the mounted file system entries that match the request criteria could not fit in the buffer space given, then the bytes available will **not** have the same value as bytes returned. To retrieve all the entries, the request should be made again using an output buffer of at least this size.

Length of output buffer
INPUT; BINARY(4), UNSIGNED

The length of the output buffer provided. The length of the output buffer parameter may be specified up to the size of the output buffer area specified by the user program. The length of the output buffer should be 0 when the output buffer is NULL.

Error code
I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

Output Buffer Description

The following tables describe the order and format of the data returned in the output buffer for each of the allowable file system operations. For a detailed description of each field, see “Field Descriptions” on page 321.

FLOP0100 Structure Description

This structure is used to return netgroup definitions taken from the /etc/netgroup file.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4), UNSIGNED	Bytes returned
4	4	BINARY(4), UNSIGNED	Bytes available
8	8	BINARY(4), UNSIGNED	Number of netgroup entries
These fields repeat for each netgroup entry.		BINARY(4), UNSIGNED	Length of netgroup entry
		BINARY(4), UNSIGNED	Length of netgroup name
		BINARY(4), UNSIGNED	Displacement to member names
		BINARY(4), UNSIGNED	Number of member names
		CHAR(*)	Netgroup name
These fields repeat for each member name in the netgroup entry.		BINARY(4), UNSIGNED	Length of member name entry
		BINARY(4), UNSIGNED	Member name status
		BINARY(4), UNSIGNED	Length of member name
		CHAR(*)	Member name

FLOP0300 Output Structure Description

This structure is used to return export entries given by an NFS server.

Offset			
Dec	Hex	Type	Field
0	0	BINARY(4), UNSIGNED	Bytes returned
4	4	BINARY(4), UNSIGNED	Bytes available
8	8	BINARY(4), UNSIGNED	Number of export entries

Offset		Type	Field
Dec	Hex		
These fields repeat for each export entry.		BINARY(4), UNSIGNED	Length of export entry
		BINARY(4), UNSIGNED	Length of export name
		BINARY(4), UNSIGNED	CCSID of export name
		BINARY(4), UNSIGNED	Displacement to export items
		BINARY(4), UNSIGNED	Number of export items
		CHAR(*)	Export name
These fields repeat for each export item in the export entry.		BINARY(4), UNSIGNED	Length of export item entry
		BINARY(4), UNSIGNED	Length of export item
		BINARY(4), UNSIGNED	CCSID of export item
		CHAR(*), UNSIGNED	Export item

FLOP0400 Output Structure Description

This structure is used to return mounted file system entries.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4), UNSIGNED	Bytes returned
4	4	BINARY(4), UNSIGNED	Bytes available
8	8	BINARY(4), UNSIGNED	Number of mount entries

Offset		Type	Field
Dec	Hex		
These fields repeat for each mount entry.		BINARY(4), UNSIGNED	Length of mount entry
		BINARY(8), UNSIGNED	File system id
		BINARY(4), UNSIGNED	File system type
		BINARY(4), UNSIGNED	Mount flags
		BINARY(4), UNSIGNED	Unique mount id
		BINARY(4)	Time of mount
		BINARY(4), UNSIGNED	Mount visibility
		BINARY(4), UNSIGNED	Displacement to mounted file system (MFS) name
		BINARY(4), UNSIGNED	Length of MFS name
		BINARY(4), UNSIGNED	CCSID of MFS name
		BINARY(4), UNSIGNED	Displacement to mount over dir name
		BINARY(4), UNSIGNED	Length of mount over dir name
		BINARY(4), UNSIGNED	CCSID of mount over dir name
		BINARY(4), UNSIGNED	Displacement to remote host name
		BINARY(4), UNSIGNED	Length of remote host name
		BINARY(4), UNSIGNED	CCSID of remote host name
		BINARY(4), UNSIGNED	Displacement to mount options
		BINARY(4), UNSIGNED	Length of mount options
		BINARY(4), UNSIGNED	CCSID of mount options
		CHAR(*)	MFS name
		CHAR(*)	Mount over dir name
		CHAR(*)	Remote host name
		CHAR(*)	Mount options

Input Buffer Description

The following tables describe the order and format of the data given in the input buffer parameter for each of the allowable file system operations. For a detailed description of each field, see “Field Descriptions” on page 321.

Format of FLOP0200 Structure

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4), UNSIGNED	Number of netgroup entries
These fields repeat for each netgroup entry.		BINARY(4), UNSIGNED	Length of netgroup entry
		BINARY(4), UNSIGNED	Length of netgroup name
		BINARY(4), UNSIGNED	Displacement to member names
		BINARY(4), UNSIGNED	Number of member names
		CHAR(*)	Netgroup name

Offset		Type	Field
Dec	Hex		
These fields repeat for each member name in the netgroup entry.		BINARY(4), UNSIGNED	Length of member name entry
		BINARY(4), UNSIGNED	Member name status
		BINARY(4), UNSIGNED	Length of member name
		CHAR(*)	Member name

Format of FLOP0300 Input Structure

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4), UNSIGNED	Preferred output CCSID
4	4	BINARY(4), UNSIGNED	Expected CCSID
8	8	BINARY(4), UNSIGNED	Length of server name
12	C	BINARY(4), UNSIGNED	CCSID of Server name
16	10	CHAR(256)	Server name

Format of FLOP0400 Input Structure

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4), UNSIGNED	Preferred output CCSID
4	4	BINARY(4), UNSIGNED	File system type filter
8	8	BINARY(4), UNSIGNED	Only visible mounts

Field Descriptions

Bytes available. The number of bytes of data available to be returned to the user in the output buffer. If all data is returned, bytes available is the same as the number of bytes returned. If the receiver variable was not large enough to contain all of the data, this value is set based on the total number of entries that could be returned.

Bytes returned. The number of bytes of data returned to the user in the output buffer.

CCSID of export item. The CCSID of the export item data. This may not be the same as the Preferred output CCSID if the data cannot be converted to that CCSID.

CCSID of export name. The CCSID of the export name data. This may not be the same as the Preferred output CCSID if the data cannot be converted to that CCSID.

CCSID of MFS name. The CCSID of the MFS name data. This may not be the same as the Preferred output CCSID if the data cannot be converted to that CCSID.

CCSID of mount options. The CCSID of the Mount options data. This may not be the same as the Preferred output CCSID if the data cannot be converted to that CCSID.

CCSID of mount over dir name. The CCSID of the mount over dir name data. This may not be the same as the Preferred output CCSID if the data cannot be converted to that CCSID.

CCSID of remote host name name. The CCSID of the remote host name data. This may not be the same as the Preferred output CCSID if the data cannot be converted to that CCSID.

CCSID of server name. The CCSID of the server name. A value of 0 indicates that the data is in the CCSID of the job.

Displacement to export items. The offset (in bytes) from the beginning of the export entry to the export items in the export entry.

Displacement to member names. The offset (in bytes) from the beginning of the netgroup entry to the member names in the netgroup entry.

Displacement to MFS name. The offset (in bytes) from the beginning of the mount entry to the mounted file system (MFS) name in the entry.

Displacement to mount options. The offset (in bytes) from the beginning of the mount entry to the mount options in the entry.

Displacement to mount over dir name. The offset (in bytes) from the beginning of the mount entry to the mount over dir name in the entry.

Displacement to remote host name. The offset (in bytes) from the beginning of the mount entry to the remote host name in the entry. If the value is 0, then there is no remote host name associated with the mount entry.

Expected CCSID. This value should contain the CCSID that the remote NFS server is expected to return string data in. A value of 0 means to calculate an ASCII CCSID based on the default CCSID of the job (recommended).

Export item. Information item that pertains to the current export. Export items are controlled by the NFS server, and it is not specified what they will contain. They are assumed to be strings and are converted into the Preferred output CCSID, if possible. Normally, an export item contains the hostname of a machine allowed to access or mount the export.

Export name. The pathname of the returned export.

File system id. A number uniquely identifying the mounted file system. Each returned mount entry should have a different file system id.

File system type. Identifies the type of the mounted file system. Refer to the different type values given under the file system type filter field description below.

File system type filter. An ORed value of flags to limit the types of mounted file systems to return. It must be a combination of the following file system type values:

File System Type Value (Hex)	File System Type Value (Integer)	File System Type
0x00000000	0	Other (Non-Specified)
0x00000001	1	"Root" (/)
0x00000002	2	QOpenSys
0x00000004	4	QDLS

File System Type Value (Hex)	File System Type Value (Integer)	File System Type
0x00000008	8	QSYS.LIB
0x00000010	16	NFS Version 2
0x00000020	32	NFS Version 3
0x00000040	64	User-Defined File System (UDFS)
0x00000080	128	Optical
0x00000100	256	QFileServer.400
0x00000200	512	Netware
0x00000400	1024	QNTC
0x00000800	2048	Independent ASP QSYS.LIB
0x00001000	4096	UDFS Management
0x00000270	624	All Dynamic MFS
0xFFFFFFFF	4294967295	All MFS

Note: All Dynamic MFS includes all of the dynamically mounted file systems: Network File System (NFS), User-Defined File Systems (UDFS), and Netware. These file systems can be mounted on demand in different parts of the namespace.

Length of export entry. The length (in bytes) of the current export entry. The length can be used to access the next entry.

Length of export item. The length (in bytes) of the export item.

Length of export item entry. The length (in bytes) of the current export item entry. The length can be used to access the next entry.

Length of export name. The length (in bytes) of the exported name (export pathname).

Length of member name. The length (in bytes) of the member name.

Length of member name entry. The length (in bytes) of this member name entry.

Length of MFS name. The length (in bytes) of the mounted file system name.

Length of mount entry. The length (in bytes) of the current mount entry. The length can be used to access the next entry.

Length of mount options. The length (in bytes) of the mount options.

Length of mount over dir name. The length (in bytes) of the mount over dir name.

Length of netgroup entry. The length (in bytes) of the current netgroup entry. The length can be used to access the next entry.

Length of netgroup name. The length (in bytes) of the netgroup name.

Length of remote host name. The length (in bytes) of the remote host name. This value will be 0 when the file system is not mounted from a remote host.

Length of server name. The length (in bytes) of the requested server name which follows. The maximum value for this field is 255.

Member name. The member name. This is assumed to be in the CCSID of the job.

Member name status. Describes the type of member name. Possible values follow:

(1) QP0L_MEMBER_IS_A_HOST_NAME
The member name refers to an individual host name.

(2) QP0L_MEMBER_IS_A_NETGROUP_NAME
The member name refers to a netgroup name.

(3) QP0L_MEMBER_IS_AN_IP_ADDRESS
The member name refers to an IP address in the form xxx.xxx.xxx.xxx (for example 123.4.56.78).

MFS name. The name of the mounted file system. This is normally the source path name.

Mount flags. An ORed value of flags that supplies information on how the file system is mounted.

Mount Flag Value	Mount Flag Description
0x0001	File system is read-only
0x0002	File system is not case sensitive
0x0004	Renaming of a file to a different casing of the same name will change the casing of the name
0x0008	File system cannot be mounted over
0x0010	File system cannot be exported through NFS
0x0020	File system can be dynamically unmounted
0x0040	File system supports synchronous writes
0x0080	File system is thread safe
0x0100	Default file format for *STMF objects is *TYPE1
0x0200	File system supports the SUID and SGID mode bits, but the bits are not surfaced due to a mount option
0x0400	File system is a Network File System hard mount

Mount options. The string representation of the valid options used to mount the file system. Valid options vary by the type of the mounted file system.

Mount over dir name. The pathname of the directory that is mounted over by the mounted file system. This is where the mount is accessible in the local system's namespace if the mounted file system is visible.

Mount visibility. A value of 1 indicates this mount has **not** been mounted over and is accessble (visible) through the parent file system's namespace. A value of 0 indicates the mounted file system has itself been mounted over.

Netgroup name. The netgroup name. This is assumed to be in the CCSID of the job.

Number of export entries. The number of complete export entries returned. A value of zero is used if there are no exports available on the server or if insufficient space was provided to hold even a single entry.

Number of export items. The number of export items for this export entry.

Number of member names. The number of member names in the netgroup entry.

Number of mount entries. The number of complete mounted file system entries returned. A value of zero is used if there are no mounts meeting the selection criteria or if insufficient space was provided to hold even a single entry.

Number of netgroup entries. The number of complete entries. A value of zero is used if there are no valid entries for the /etc/netgroup file or if the file does not exist.

Only visible mounts. A value of 1 requests that only visible (accessible, topmost) mounted file systems be retrieved. A value of 0 means to not limit the retrieved mounts based on visiblity.

Preferred output CCSID. The CCSID into which the output will be converted. If a conversion failure occurs, the output may be returned in another CCSID. A value of 0 indicates that the data should be returned in the CCSID of the job.

Remote host name. The name of the host on which the source file system resides. This is the machine being mounted from and is only applicable for remote mounts. For local mounts, the value of Displacement to remote host name will be 0, and this value will not be returned.

Server name. The host name of the server to retrieve the Network File System (NFS) export entries from.

Time of mount. The time when the file system was mounted.

Unique mount id. This value gives an indication of the order in which the file systems were mounted. For example, multiple file systems may be mounted over the same directory. The topmost one (and therefore the one that is visible) will be the one with the largest mount sequence number. » The mount sequence numbers will be reset after any system processing which unmounts and mounts file systems, such as IPL and Reclaim Storage (RCLSTG). This value corresponds to the value returned by "stat()—Get File Information" on page 468 and similar APIs in the st_vfs field. «

Usage Notes

1. The include file for this API is QP0LFLOP.
2. If none of the required parameters are passed to this API, then message CPFB41F will be issued to the caller. This message lists all of the file operations currently available to the QP0LFLOP API.
3. *WARNING* - When the (2) QP0L_WRITE_NETGROUP_FILE_ENTRIES file system operation is requested, the existing /etc/netgroup file will be completely rewritten resulting in a loss of the previous contents of the file.
4. A netgroup is a way of defining one name (the netgroup name) to represent many other names. The names contained within a netgroup definition are called 'members' of that netgroup. A netgroup member can be either the name of a host system, the name of another netgroup, or an IP address. Netgroup definitions are stored in the /etc/netgroup file and are commonly used by the Network File System (NFS) support when a large group of host systems require common NFS access semantics.
5. An export entry describes a remote file system or subdirectory in a file system residing on an Network File System (NFS) server that is mountable by an NFS client.

Error Messages

Message ID	Error Message Text
CPFA0D4 E	File system error occurred.
CPE3418 E	Possible APAR condition or hardware failure.
CPF3C90 E	Literal value cannot be changed.
CPF3CF1 E	Error code parameter not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.

Message ID	Error Message Text
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPDA1B9 E	An error has occurred in the Network File System (NFS).
CPFA0AA E	Error occurred while attempting to obtain space.
CPFA0D0 E	CCSID conversion error occurred.
CPFA1CE E	Cannot find address for specified system name.
CPFB41F E	File system operation failed.

API introduced: V4R3

Top | UNIX-Type APIs | APIs by category

Qp0lGetAttr()—Get Attributes

Syntax

```
#include <Qp0lstdi.h>
int Qp0lGetAttr
  (Qlg_Path_Name_T        *Path_Name,
   Qp0l_AttrTypes_List_t  *Attr_Array_ptr,
   char                   *Buffer_ptr,
   uint                    Buffer_Size_Provided,
   uint                   *Buffer_Size_Needed_ptr,
   uint                   *Num_Bytes_Returned_ptr,
   uint                    Follow_Symlnk, ...);
```

Service Program Name: QP0LLIB2
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 347.

The **Qp0lGetAttr()** function gets one or more attributes, on a single call, for the object that is referred to by the input *Path_Name*. The object must exist, the user must have authority to it, and the requested attributes must be supported by the specific file system or object type. For each requested attribute that is not supported by the file system or object type, **Qp0lGetAttr()** returns zero in the Size of attribute data field, pointed to by the *Buffer_ptr* parameter, for that attribute.

Qp0lGetAttr() either returns the attributes of the symbolic link, or returns the attributes of the object that the symbolic link names. This depends upon the value of the *Follow_Symlnk* parameter.

Qp0lGetAttr() returns all times in seconds since the Epoch so that they are consistent with UNIX-type APIs. The Epoch is the time 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time. If the i5/OS date is set prior to 1970, all time values are zero.

Parameters

Path_Name
: (Input) The path name of the object for which attribute information is returned. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path name format.

Attr_Array_ptr
: (Input) A pointer to a structure listing the requested attributes returned for the object identified by the *Path_Name* parameter. Each entry in the array identifies an attribute, by a constant value, that **Qp0lGetAttr()** returns. The number of requested attributes field must equal the total number of constants. If the **Attr_Array_ptr** is NULL or if the Number of requested attributes field is zero, **Qp0lGetAttr()** returns all the attributes that the API supports that are available for the object. The format of this parameter follows.

Attribute array pointer			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Number of requested attributes
4	4	ARRAY(*) of BINARY(4)	Array of attribute constants

Array of attribute constants. A list of predefined constants, each identifying a requested attribute. **Qp0lGetattr()** also returns one of these constants in the Attribute identification field, pointed to by the *Buffer_ptr* parameter. The constant must be used to identify the returned attribute because the attributes are returned in any order. Note that the Size of attribute data field, pointed to by the *Buffer_ptr* parameter, contains the total size of data that **Qp0lGetattr()** returns for the constants in this array. Valid values, and sizes of the returned attributes, follow:

0 QP0L_ATTR_OBJTYPE: (CHAR(10)) The object type. See Control Language (CL) information in the iSeries Information center for descriptions of all iSeries object types.

1 QP0L_ATTR_DATA_SIZE: (UNSIGNED BINARY(4)) The size in bytes of the data in this object. The size varies by object type and file system. This size does not include object headers or the size of extended attributes associated with the object. If this attribute is requested and the size cannot be represented in a BINARY(4) data type, **Qp0lGetAttr()** fails with *errno* [EOVERFLOW]. Refer to QP0L_ATTR_DATA_SIZE_64 for objects whose data sizes are greater than BINARY(4).

2 QP0L_ATTR_ALLOC_SIZE: (UNSIGNED BINARY(4)) The number of bytes that have been allocated for this object. The allocated size varies by object type and file system. For example, the allocated size includes the object data size as shown in QP0L_ATTR_DATA_SIZE or QP0L_ATTR_DATA_SIZE_64 as well as any logically sized extents to accommodate anticipated future requirements for the object data. It may or may not include additional bytes for attribute information. If this size cannot be represented in a BINARY(4) data type, **Qp0lGetAttr()** fails with *errno* [EOVERFLOW]. Refer to QP0L_ATTR_ALLOC_SIZE_64 for objects whose allocated sizes are greater than BINARY(4).

3 QP0L_ATTR_EXTENDED_ATTR_SIZE: (UNSIGNED BINARY(4)) The total number of extended attribute bytes.

4 QP0L_ATTR_CREATE_TIME: (UNSIGNED BINARY(4)) The time the object was created.

5 QP0L_ATTR_ACCESS_TIME: (UNSIGNED BINARY(4)) The time that the object's data was last accessed.

6 QP0L_ATTR_CHANGE_TIME: (UNSIGNED BINARY(4)) The time that the object's data or attributes were last changed.

7 QP0L_ATTR_MODIFY_TIME: (UNSIGNED BINARY(4)) The time that the object's data was last changed.

8 QP0L_ATTR_STG_FREE: (CHAR(1)) Whether the object's data has been moved offline, freeing its online storage. Valid values are:

x'00' QP0L_SYS_NOT_STG_FREE: The object's data is not offline.
x'01' QP0L_SYS_STG_FREE: The object's data is offline.

9 QP0L_ATTR_CHECKED_OUT: Whether an object is checked out or not. When an object is checked out, other users can read and copy the object. Only the user who has the object checked out can change the object. The checkout format is defined in the

Qp0lstdi.h header file as data type Qp0l_Checkout_t, and is described in the following table.

Checkout Format			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(1)	Flag indicating whether an object is checked out
1	1	CHAR(10)	User to whom checked out
11	B	CHAR(1)	Reserved
12	C	BINARY(4)	Time checked out

Flag. An indicator as to whether an object is checked out. Valid values are:

x'00' QP0L_NOT_CHECKED_OUT: The object is not checked out.
x'01' QP0L_CHECKED_OUT: The object is checked out.

Reserved. A reserved field. This field must be set to binary zero.

Time checked out. The time the object was checked out. This field represents the number of seconds since the Epoch.

User to whom checked out. The user who has the object checked out. This field is blank if it is not checked out.

10 QP0L_ATTR_LOCAL_REMOTE: (CHAR(1)) Whether an object is stored locally or stored on a remote system. The decision of whether a file is local or remote varies according to the respective file system rules. Objects in file systems that do not carry either a local or remote indicator are treated as remote. Valid values are:

x'01' QP0L_LOCAL_OBJ: The object's data is stored locally.
x'02' QP0L_REMOTE_OBJ: The object's data is on a remote system.

11 QP0L_ATTR_AUTH: The public and private authorities associated with the object.

When the QP0L_ATTR_AUTH attribute is requested, the attribute data is returned in the buffer in the following format. This format is defined in header file Qp0lstdi.h as data type Qp0l_Authority_General_t.

General Authority Format			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(10)	Object owner
10	0A	CHAR(10)	Primary group
20	14	CHAR(10)	Authorization list name
30	1E	CHAR(10)	Reserved
40	28	BINARY(4)	Offset to array of users
44	2C	BINARY(4)	Number of users
48	30	BINARY(4)	Size of user entry field entry
52	34	CHAR(12)	Reserved
		ARRAY(*)	Array of users

Array of users. The names and authorities of the users who are authorized to use the object.

Authorization list name. The name of the authorization list that is used to secure the named object. The value *NONE indicates that no authorization list is used in determining authority to the object.

Number of users. The number of users that are authorized to the object. This is the number of users returned in the array of users.

The QFileSvr.400 file system returns zero for the Number of users and zero for the Offset to array of users. If a primary group is specified, the Network File System (NFS) returns one for the Number of users.

Object owner. The name of the user profile that is the owner of the object or the following special value:

**NOUSRPRF* This special value is used by the Network File System to indicate that there is no user profile on the local iSeries server with a user ID (UID) matching the UID of the remote object.

Offset to array of users. The offset to the names and authorities of the users who are authorized to use the object. This offset is relative to the start of the buffer pointed to by the *Buffer_ptr* parameter.

Primary group. The name of the user profile that is the primary group of the object or the following special values:

**NONE* The object does not have a primary group.

**NOUSRPRF* This special value is used by the Network File System to indicate that there is no user profile on the local server with a group ID (GID) matching the GID of the remote object.

Reserved. A reserved field. This field must be set to binary zero.

Size of user entry field entry. The number of bytes returned for each user.

When the QP0L_ATTR_AUTH attribute is requested, the array of users is returned in the buffer in the following format. This format is defined in header file Qp0lstdi.h as data type Qp0l_Authority_Users_t.

Data and Object Authority Format			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(10)	User name
10	0A	CHAR(10)	User data authority
Object rights			
20	14	CHAR(1)	Object management
21	15	CHAR(1)	Object existence
22	16	CHAR(1)	Object alter
23	17	CHAR(1)	Object reference
24	18	CHAR(10)	Reserved
Data rights			
34	22	CHAR(1)	Object operational
35	23	CHAR(1)	Read
36	24	CHAR(1)	Add

Data and Object Authority Format			
Offset			
Dec	Hex	Type	Field
37	25	CHAR(1)	Update
38	26	CHAR(1)	Delete
39	27	CHAR(1)	Execute
40	28	CHAR(1)	Exclude
41	29	CHAR(7)	Reserved

Add (*ADD). Authority to add entries to the object. Valid values are:

0 The user does not have add data rights.
1 The user does have add data rights.

Delete (*DELETE). Authority to remove entries from the object. Valid values are:

0 The user does not have delete data rights.
1 The user does have delete data rights.

Execute (*EXECUTE). Authority to run a program or search a library or directory. Valid values are:

0 The user does not have execute data rights.
1 The user does have execute data rights.

Exclude (*EXCLUDE). The user is prevented from accessing the object. Valid values are:

0 The user does not have exclude data rights.
1 The user does have exclude data rights.

Object alter (*OBJALTER). Authority to change the attributes of an object, such as adding or removing triggers for a database file. Valid values are:

0 The user does not have alter object rights.
1 The user does have alter object rights.

Object existence (*OBJEXIST). Authority to control the object's existence and ownership. Valid values are:

0 The user does not have object existence rights.
1 The user does have object existence rights.

Object management (*OBJMGT). Authority to specify security, to move or rename the object, and to add members if the object is a database file. Valid values are:

0 The user does not have object management rights.
1 The user does have object management rights.

Object operational (*OBJOPR). Authority to look at the object's attributes and to use the object as specified by the data authorities that the user has to the object. Valid values are:

0 The user does not have object operational rights.

1 The user does have object operational rights.

Object reference (*OBJREF). Authority to specify the object as the first level in a referential constraint. Valid values are:

0 The user does not have object reference rights.
1 The user does have object reference rights.

Read (*READ). Authority to access the contents of the object. Valid values are:

0 The user does not have read data rights.
1 The user does have read data rights.

Reserved. A reserved field. This field must be set to binary zero.

Update (*UPDATE). Authority to change the content of existing entries in the object. Valid values are:

0 The user does not have update data rights.
1 The user does have update data rights.

User data authority. The operation, use, or access that the user has to an object. Valid values follow:

**RWX* Allows all operations on the object except those that are limited to the owner or controlled by the object rights.
**RW* Allows access to the object attributes and allows the object to be changed. The user cannot use the object.
**WX* Allows use of the object and allows the object to be changed. The user cannot access the object attributes.
**R* Allows access to the object attributes.
**W* Allows the object to be changed.
**X* Allows the use of the object.
**EXCLUDE* All operations on the object are prohibited.
**NONE* Displayed by the system when the user does not have any data authorities.
USER DEF Displayed by the system when the specific data authorities do not match any of the predefined data authority levels above.

User name. The name of a user authorized to use the object. This may be the name of the user profile or one of the following special values:

**NOUSRPRF* The authorities of either the owner or the primary group of the object for which the profile name could not be determined. This value is used by the Network File System only. It indicates that the user ID (UID) or the group ID (GID) for the remote object does not match any profile on the local iSeries server with that UID or GID.
**NTWIRF* The authorities of the NetWare Inherited Rights Filter for the object. This value is only used by the QNetWare file system.
**NTWEFF* The NetWare effective rights to the object. This value is only used by the QNetWare file system.
**PUBLIC* The authorities of users who are not specifically named and who are not in the object's authorization list.

12 QP0L_ATTR_FILE_ID: (CHAR(16)) An identifier associated with the referred to object. A file ID can be used with "Qp0lGetPathFromFileID()—Get Path Name of Object from Its File ID" on page 351 to retrieve an object's path name. The file ID is defined in header file Qp0lstdi.h as data type Qp0lFID_t.

13 QP0L_ATTR_ASP: (BINARY(2)) The auxiliary storage pool in which the object is stored.

14 QP0L_ATTR_DATA_SIZE_64: (UNSIGNED BINARY(8)) The size in bytes of the data in this object. The size varies by object type and file system. This size does not include object headers or the size of extended attributes associated with the object. QP0L_ATTR_DATA_SIZE may be used for objects whose data size can be represented in a BINARY(4) data type.

15 QP0L_ATTR_ALLOC_SIZE_64: (UNSIGNED BINARY(8)) The number of bytes that have been allocated for this object. The allocated size varies by object type and file system. For example, the allocated size includes the object data size as shown in QP0L_ATTR_DATA_SIZE or QP0L_ATTR_DATA_SIZE_64 as well as any logically sized extents to accommodate anticipated future requirements for the object data. It may or may not include additional bytes for attribute information. QP0L_ATTR_ALLOC_SIZE may be used for objects whose allocated size can be represented in a BINARY(4) data type.

16 QP0L_ATTR_USAGE_INFORMATION: Fields indicating how often an object is used. Usage has different meanings according to the specific file system and according to the individual object types supported within a file system. Usage can indicate the opening or closing of a file or can refer to adding links, renaming, restoring, or checking out an object. The usage information format is defined in the Qp0lstdi.h header file as data type Qp0l_Usage_t and is shown in the following table.

Qp0l_Usage_t			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Reset date
4	4	BINARY(4)	Last used date
8	8	BINARY(4)	Days used count
12	C	CHAR(4)	Reserved

Days used count. The number of days an object has been used. Usage has different meanings according to the specific file system and according to the individual object types supported within a file system. Usage can indicate the opening or closing of a file or can refer to adding links, renaming, restoring, or checking out an object. This count is incremented once each day that an object is used and is reset to zero by calling the **Qp0lSetAttr()** API.

Last used date. The number of seconds since the Epoch that corresponds to the date the object was last used. This field is zero when the object is created. If usage data is not maintained for the i5/OS type or the file system to which an object belongs, this field is zero.

Reserved. A reserved field set to binary zeros.

Reset date. The number of seconds since the Epoch that corresponds to the date the days used count was last reset to zero (0). This date is set to the current date when the **Qp0lSetAttr()** API is called to reset the Days used count to zero.

17 QP0L_ATTR_PC_READ_ONLY: (CHAR(1)) Whether the object can be written to or deleted, have its extended attributes changed or deleted, or have its size changed. Valid values are:

x'00' QP0L_PC_NOT_READONLY: The object can be changed.
x'01' QP0L_PC_READONLY: The object cannot be changed.

18 QP0L_ATTR_PC_HIDDEN: (CHAR(1)) Whether the object can be displayed using an ordinary directory listing.

x'00' QP0L_PC_NOT_HIDDEN: The object is not hidden.
x'01' QP0L_PC_HIDDEN: The object is hidden.

19 QP0L_ATTR_PC_SYSTEM: (CHAR(1)) Whether the object is a system file and is excluded from normal directory searches.

x'00' QP0L_PC_NOT_SYSTEM: The object is not a system file.
x'01' QP0L_PC_SYSTEM: The object is a system file.

20 QP0L_ATTR_PC_ARCHIVE: (CHAR(1)) Whether the object has changed since the last time the file was examined.

x'00' QP0L_PC_NOT_CHANGED: The object has not changed.
x'01' QP0L_PC_CHANGED: The object has changed.

21 QP0L_ATTR_SYSTEM_ARCHIVE: (CHAR(1)) Whether the object has changed and needs to be saved. It is set on when an object's change time is updated, and set off when the object has been saved.

x'00' QP0L_SYSTEM_NOT_CHANGED: The object has not changed and does not need to be saved.
x'01' QP0L_SYSTEM_CHANGED: The object has changed and does need to be saved.

22 QP0L_ATTR_CODEPAGE: (BINARY(4)) The code page derived from the coded character set identifier (CCSID) used for the data in the file or the extended attributes of the directory. If the returned value of this field is zero (0), there is more than one code page associated with the st_ccsid. If the st_ccsid is not a supported system CCSID, the st_codepage is set equal to the st_ccsid.

23 QP0L_ATTR_FILE_FORMAT: (CHAR(1)) The format of the stream file (*STMF). Valid values are:

x'00' QP0L_FILE_FORMAT_TYPE1: The object has the same format as *STMF objects created on releases prior to Version 4 Release 4. It has a minimum object size of 4096 bytes and a maximum object size of approximately » 128 « gigabytes.
x'01' QP0L_FILE_FORMAT_TYPE2: A QP0L_FILE_FORMAT_TYPE2 (*TYPE2) *STMF has high performance file access and was new in Version 4 Release 4 of i5/OS (OS/400). It has a minimum object size of 4096 bytes and a maximum object size of approximately one terabyte in the "root" (/), QOpenSys and user-defined file systems. Otherwise, the maximum is approximately 256 gigabytes. A *TYPE2 *STMF is capable of memory mapping as well as the ability to specify an attribute to optimize disk storage allocation.

24 QP0L_ATTR_UDFS_DEFAULT_FORMAT: (CHAR(1)) The default file format of stream files (*STMF) created in the user-defined file system. Valid values are:

x'00' QP0L_UDFS_DEFAULT_TYPE1: The stream file (*STMF) has the same format as *STMFs created on releases prior to Version 4 Release 4 of i5/OS (OS/400). It has a minimum object size of 4096 bytes and a maximum object size of approximately 256 gigabytes.
x'01' QP0L_UDFS_DEFAULT_TYPE2: A *TYPE2 *STMF has high performance file access and was new in Version 4 Release 4 of i5/OS (OS/400). It has a minimum object size of 4096 bytes and a maximum object size of approximately one terabyte in the "root" (/), QOpenSys and user-defined file systems. Otherwise, the maximum is approximately 256 gigabytes. A *TYPE2 *STMF is capable of memory mapping as well as the ability to specify an attribute to optimize disk storage allocation.

25 QP0L_ATTR_JOURNAL_INFORMATION: Basic Journaling information for this object.

The journaling information format is defined in the Qp0lstdi.h header file as data type Qp0l_Journal_Info_t and is shown in the following table:

Qp0l_Journal_Info_t			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(1)	Journaling status
1	1	CHAR(1)	Options
2	2	CHAR(10)	Journal identifier (JID)
12	0C	CHAR(10)	Current or last journal name
22	16	CHAR(10)	Current or last journal library name
32	20	BINARY(4), UNSIGNED	Last journaling start time

For extended journaling information see QP0L_ATTR_JOURNAL_EXTENDED_INFORMATION.

Current or last journal library name. If the value of the journaling status is QP0L_JOURNALED, then this field contains the name of the library containing the currently used journal. If the value of the journaling status is QP0L_NOT_JOURNALED, then this field contains the name of the library containing the last used journal. All bytes in this field will be set to binary zero if this object has never been journaled.

Current or last journal name. If the value of the journaling status is QP0L_JOURNALED, then this field contains the name of the journal currently being used. If the value of the journaling status is QP0L_NOT_JOURNALED, then this field contains the name of the journal last used for this object. All bytes in this field will be set to binary zero if this object has never been journaled.

Journal identifier (JID). This field associates the object being journaled with an identifier that can be used on various journaling-related commands and APIs.

Journaling status. Current journaling status of the object. This field will be one of the following values:

x'00' QP0L_NOT_JOURNALED: The object is currently not being journaled.

x'01' QP0L_JOURNALED: The object is currently being journaled.

Last journaling start time. The number of seconds since the Epoch that corresponds to the last date and time for which the object had journaling started for it. This field will be set to binary zero if this object has never been journaled.

Options. This field describes the current journaling options. This field is composed of several bit flags and contains one or more of the following bit values:

x'80' QP0L_JOURNAL_SUBTREE: When this flag is returned, this object is a directory with IFS journaling subtree semantics. New objects created within this directory's subtree will inherit the journaling attributes and options from this directory.

x'08' QP0L_JOURNAL_OPTIONAL_ENTRIES: When journaling is active, entries that are considered optional are journaled. The list of optional journal entries varies for each object type. See the Integrated file system information in the Files and file systems topic for information regarding these optional entries for various objects.

x'20' QP0L_JOURNAL_AFTER_IMAGES: When journaling is active, the image of the object after a change is journaled.

x'40' QP0L_JOURNAL_BEFORE_IMAGES: When journaling is active, the image of the object prior to a change is journaled.

26 QP0L_ATTR_ALWCKPWRT: (CHAR(1)) Whether a stream file (*STMF) can be shared with readers and writers during the save-while-active checkpoint processing. Valid values are:

x'00' QP0L_NOT_ALWCKPWRT: The object can be shared with readers only.
x'01' QP0L_ALWCKPWRT: The object can be shared with readers and writers.

27 QP0L_ATTR_CCSID: (BINARY(4)) The CCSID of the data and extended attributes of the object.

28 QP0L_ATTR_SIGNED: (CHAR(1)) Whether an object has an i5/OS digital signature. This attribute is only returned for *STMF objects. Valid values are:

x'00' QP0L_NOT_SIGNED: The object does not have an i5/OS digital signature.
x'01' QP0L_SIGNED: The object does have an i5/OS digital signature.

29 QP0L_ATTR_SYS_SIGNED: (CHAR(1)) Whether the object was signed by a source that is trusted by the system. This attribute is only returned for *STMF objects. Note: this attribute is not returned if the QP0L_ATTR_SIGNED attribute has the value QP0L_NOT_SIGNED. Valid values are:

x'00' QP0L_SYSTEM_SIGNED_NO: (CHAR(1)) None of the signatures came from a source that is trusted by the system.
x'01' QP0L_SYSTEM_SIGNED_YES: The object was signed by a source that is trusted by the system. If the object has multiple signatures, at least one of the signatures came from a source that is trusted by the system.

30 QP0L_ATTR_MULT_SIGS: (CHAR(1)) Whether an object has more than one i5/OS digital signature. This attribute is only returned for *STMF objects. Note: this attribute is not returned if the QP0L_ATTR_SIGNED attribute has the value QP0L_NOT_SIGNED. Valid values are:

x'00' QP0L_MULT_SIGS_NO: The object has only one digital signature.
x'01' QP0L_MULT_SIGS_YES: The object has more than one digital signature. If the QP0L_ATTR_SYS_SIGNED attribute has the value QP0L_SYS_SIGNED, at least one of the signatures is from a source trusted by the system.

31 QP0L_ATTR_DISK_STG_OPT (CHAR(1)) This option should be used to determine how auxiliary storage is allocated by the system for the specified object. This option can only be specified for stream files in the "root" (/), QOpenSys and user-defined file systems. This option will be ignored for *TYPE1 byte stream files. Valid values are:

x'00' QP0L_STG_NORMAL: The auxiliary storage will be allocated normally. That is, as additional auxiliary storage is required, it will be allocated in logically sized extents to accommodate the current space requirement, and anticipated future requirements, while minimizing the number of disk I/O operations.
x'01' QP0L_STG_MINIMIZE: The auxiliary storage will be allocated to minimize the space used by the object. That is, as additional auxiliary storage is required, it will be allocated in small sized extents to accommodate the current space requirement. Accessing an object composed of many small extents may increase the number of disk I/O operations for that object.

x'02' QP0L_STG_DYNAMIC: The system will dynamically determine the optimum auxiliary storage allocation for the object, balancing space used versus disk I/O operations. For example, if a file has many small extents, yet is frequently being read and written, then future auxiliary storage allocations will be larger extents to minimize the number of disk I/O operations. Or, if a file is frequently truncated, then future auxiliary storage allocations will be small extents to minimize the space used. Additionally, information will be maintained on the stream file sizes for this system and its activity. This file size information will also be used to help determine the optimum auxiliary storage allocations for this object as it relates to the other objects sizes.

32 QP0L_ATTR_MAIN_STG_OPT: (CHAR(1)) This option should be used to determine how main storage is allocated and used by the system for the specified object. This option can only be specified for stream files in the ″root″ (/), QOpenSys and user-defined file systems. Valid values are:

x'00' QP0L_STG_NORMAL: The main storage will be allocated normally. That is, as much main storage as possible will be allocated and used. This minimizes the number of disk I/O operations since the information is cached in main storage.

x'01' QP0L_STG_MINIMIZE: The main storage will be allocated to minimize the space used by the object. That is, as little main storage as possible will be allocated and used. This minimizes main storage usage while increasing the number of disk I/O operations since less information is cached in main storage.

x'02' QP0L_STG_DYNAMIC: The system will dynamically determine the optimum main storage allocation for the object depending on other system activity and main storage contention. That is, when there is little main storage contention, as much storage as possible will be allocated and used to minimize the number of disk I/O operations. And when there is significant main storage contention, less main storage will be allocated and used to minimize the main storage contention. This option only has an effect when the storage pool's paging option is *CALC. When the storage pool's paging option is *FIXED, the behavior is the same as QP0L_STG_NORMAL. When the object is accessed through a file server, this option has no effect. Instead, its behavior is the same as QP0L_STG_NORMAL.

33 QP0L_ATTR_DIR_FORMAT: (CHAR(1)) The format of the specified directory object. Valid values are:

x'00' QP0L_DIR_FORMAT_TYPE1: The directory of type *DIR has the original directory format. The Convert Directory (CVTDIR) command may be used to convert from the *TYPE1 format to the *TYPE2 format.

x'01' QP0L_DIR_FORMAT_TYPE2: The directory of type *DIR is optimized for performance, size, and reliability compared to directories having the *TYPE1 format.

34 QP0L_ATTR_AUDIT: (CHAR(10)) The auditing value associated with the object.

Valid values are:

**NONE* QP0L_AUD_NONE: No auditing occurs for this object when it is read or changed regardless of the user who is accessing the object.

**USRPRF* QP0L_AUD_USRPRF: Audit this object only if the current user is being audited. The current user is tested to determine if auditing should be done for this object. The user profile can specify if only change access is audited or if both read and change accesses are audited for this object. » The OBJAUD parameter of the Change User Auditing (CHGUSRAUD) command is used to change the auditing for a specific user. «

**CHANGE* QP0L_AUD_CHANGE: Audit all change access to this object by all users on the system.

**ALL* QP0L_AUD_ALL: Audit all access to this object by all users on the system. All access is defined as a read or change operation.

» **NOTAVL* QP0L_AUD_NOTAVL: The user performing the operation is not allowed to retrieve the current auditing value.

Note: The user must have all object (*ALLOBJ) or audit (*AUDIT) special authority to retrieve the auditing value. «

35 QP0L_ATTR_CRTOBJSCAN: (CHAR(1)) Whether the objects created in a directory will be scanned when exit programs are registered with any of the integrated file system scan-related exit points.

The integrated file system scan-related exit points are:

- "Integrated File System Scan on Close Exit Program" on page 513
- "Integrated File System Scan on Open Exit Program" on page 523.

This attribute can only have been specified for directories in the "root" (/), QOpenSys and user-defined file systems. Even though this attribute can be set for *TYPE1 and *TYPE2 directories, only objects which are in » file systems that have completely converted to the *TYPE2 directory format « will actually be scanned, no matter what value is set for this attribute.

Valid values are:

x'00' QP0L_SCANNING_NO: After an object is created in the directory, the object will not be scanned according to the rules described in the scan-related exit programs.

Note: If the Scan file systems control (QSCANFSCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore.

x'01' QP0L_SCANNING_YES: After an object is created in the directory, the object will be scanned according to the rules described in the scan-related exit programs if the object has been modified or if the scanning software has been updated since the last time the object was scanned.

x'02' QP0L_SCANNING_CHGONLY: After an object is created in the directory, the object will be scanned according to the rules described in the scan-related exit programs only if the object has been modified since the last time the object was scanned. It will not be scanned if the scanning software has been updated. This attribute only takes effect if the Scan file systems control (QSCANFSCTL) system value has *USEOCOATR specified. Otherwise, it will be treated as if the attribute is QP0L_SCANNING_YES.

Note: If the Scan file systems control (QSCANFSCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore.

36 QP0L_ATTR_SCAN: (CHAR(1)) Whether the object will be scanned when exit programs are registered with any of the integrated file system scan-related exit points.

The integrated file system scan-related exit points are:

- "Integrated File System Scan on Close Exit Program" on page 513
- "Integrated File System Scan on Open Exit Program" on page 523.

This attribute can only have been specified for stream files in the "root" (/), QOpenSys and user-defined file systems. Even though this attribute can be set for objects in *TYPE1 and *TYPE2 directories, only objects which are in » file systems that have completely converted to the *TYPE2 directory format « will actually be scanned, no matter what value is set for this attribute.

Valid values are:

x'00' QP0L_SCANNING_NO: The object will not be scanned according to the rules described in the scan-related exit programs.

Note: If the Scan file systems control (QSCANFSCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore.

x'01' QP0L_SCANNING_YES: The object will be scanned according to the rules described in the scan-related exit programs if the object has been modified or if the scanning software has been updated since the last time the object was scanned.

x'02' QP0L_SCANNING_CHGONLY: The object will be scanned according to the rules described in the scan-related exit programs only if the object has been modified since the last time the object was scanned. It will not be scanned if the scanning software has been updated. This attribute only takes effect if the Scan file systems control (QSCANFSCTL) system value has *USEOCOATR specified. Otherwise, it will be treated as if the attribute is QP0L_SCANNING_YES.

Note: If the Scan file systems control (QSCANFSCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore.

37 QP0L_ATTR_SCAN_INFO: Scan information for this object. The scan information format is defined in the qp0lstdi.h header file as data type Qp0l_Scan_Info_t and is shown in the following table:

Qp0l_Scan_Info_t			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(1)	Scan status
1	1	CHAR(1)	Scan signatures different
2	2	CHAR(1)	Binary scan
3	3	CHAR(1)	Reserved
4	4	BINARY(4), UNSIGNED	CCSID 1
8	8	BINARY(4), UNSIGNED	CCSID 2

Note: Historical information is only kept for the last two CCSIDs which have been scanned, as well as the binary scan indication.

Binary scan. This indicates if the object has been scanned in binary mode when it was previously scanned. This field will be one of the following values:

x'00' QP0L_SCAN_NO: The object was not scanned in binary mode.

x'01' QP0L_SCAN_YES: The object was scanned in binary mode. If the object scan status is QP0L_SCAN_SUCCESS, then the object was successfully scanned in binary. If the object scan status is QP0L_SCAN_FAILURE, then the object failed the scan in binary.

CCSID 1. A CCSID value that the object has been scanned in if it was previously scanned in a CCSID. If the object scan status is QP0L_SCAN_SUCCESS, then the object was successfully scanned in this CCSID. If the object scan status is QP0L_SCAN_FAILURE, then the object failed the scan in this CCSID. A value of 0 means this field does not apply.

CCSID 2. A CCSID value that the object has been scanned in if it was previously scanned in a CCSID. If the object scan status is QP0L_SCAN_SUCCESS, then the object was successfully scanned in this CCSID. If the object scan status is QP0L_SCAN_FAILURE, then this field will be 0. A value of 0 means this field does not apply.

Reserved. A reserved field. This field will be set to binary zero.

Scan signatures different. The scan signatures give an indication of the level of the scanning software support. For more information, see "Scan Key List and Scan Key Signatures" on page 530 in "Integrated File System Scan on Open Exit Program" on page 523.

When an object is in an independent ASP group, the object scan signature is compared to the associated independent ASP group scan signature. When an object is **not** in an independent ASP group, the object scan signature is compared to the global scan

signature value. This field will be one of the following values:

x'00' QP0L_SCAN_NO: The compared signatures are not different.
x'01' QP0L_SCAN_YES: The compared signatures are different.

Scan status. The scan status associated with this object. This field will be one of the following values:

x'00' QP0L_SCAN_REQUIRED: A scan is required for the object either because it has not yet been scanned by the scan-related exit programs, or because the objects data or CCSID has been modified since it was last scanned. Examples of object data or CCSID modifications are: writing to the object, directly or through memory mapping; truncating the object; clearing the object; and changing the objects CCSID attribute etc..

x'01' QP0L_SCAN_SUCCESS: The object has been scanned by a scan-related exit program, and at the time of that last scan request, the object did not fail the scan.

x'02' QP0L_SCAN_FAILURE: The object has been scanned by a scan-related exit program, and at the time of that last scan request, the object failed the scan and the operation did not complete. Once an object has been marked as a failure, it will not be scanned again until the object's scan signature is different than the global scan key signature or independent ASP group scan key signature as appropriate. Therefore, subsequent requests to work with the object will fail with a scan failure indication » if that access meets the criteria for when an object is to be scanned. Examples of requests which will fail are opening the object with more than write-only access, changing the CCSID of the object, copying the object etc.. See "Integrated File System Scan on Open Exit Program" on page 523 and "Integrated File System Scan on Close Exit Program" on page 513 for the criteria for when an object is to be scanned.

Note:

1. If scanning has been turned off using the QSCANFS system value, or if no exit programs are registered for a specific exit point, then any requests which trigger that specific exit point will return a scan failure indication.
2. If the scan attribute is set to not scan the object , then requests to work with the object will not fail with a scan failure indication. «

x'05' QP0L_SCAN_PENDING_CVN: The object is » in a file system that has not completely converted to the *TYPE2 directory format, and therefore will not be scanned until the file system is completely converted. For information on the *TYPE2 directory format, « see the Convert Directory (CVTDIR) command and the Integrated file system information in the Files and file systems topic.

x'06' QP0L_SCAN_NOT_REQUIRED: The object does not require any scanning because the object is marked to not be scanned.

38 QP0L_ATTR_ALWSAV: (CHAR(1)) Whether the object can be saved or not. Valid values are:

x'00' QP0L_ALWSAV_NO: This object will not be saved when using the Save Object (SAV) command or the QsrSave() API.

Additionally, if this object is a directory, none of the objects in the directory's subtree will be saved unless they were explicitly specified as an object to be saved. The subtree includes all subdirectories and the objects within those subdirectories.

Note: If this attribute is chosen for an object that has private authorities associated with it, or is chosen for the directory of an object that has private authorities associated with it, then the following consideration applies. When the private authorities are saved, the fact that an object has the QP0L_ALWSAV_NO attribute is not taken into consideration. (Private authorities can be saved using either the Save System (SAVSYS) or Save Security Data (SAVSECDTA) command or the Save Object List (QSRSAVO) API.) Therefore, when a private authority is restored using the Restore Authority (RSTAUT) command, message CPD3776 will be seen for each object that was not saved either because it had the QP0L_ALWSAV_NO attribute specified, or because the object was not specified on the save and it was in a directory that had the QP0L_ALWSAV_NO attribute specified.

x'01' QP0L_ALWSAV_YES: This object will be saved when using the Save Object (SAV) command or the QsrSave() API.

39 QP0L_ATTR_RSTDRNMUNL: (CHAR(1)) Restricted renames and unlinks for objects within a directory. Objects can be linked into a directory that has this attribute set on, but cannot be renamed or unlinked from it unless one or more of the following are true for the user performing the operation:

- The user is the owner of the object.
- The user is the owner of the directory.
- The user has *ALLOBJ special authority.

This restriction only applies to directories. Other types of object can have this attribute on, however, it will be ignored. This attribute is equivalent to the S_ISVTX mode bit for an object. Valid values are:

x'00' QP0L_RSTDRNMUNL_OFF: No additional restrictions for rename and unlink operations.
x'01' QP0L_RSTDRNMUNL_ON: Additional restrictions for rename and unlink operations.

40 QP0L_ATTR_JOURNAL_EXTENDED_INFORMATION: Extended Journaling information for this object. The journaling information format is defined in the Qp0lstdi.h header file as data type Qp0l_Journal_Extended_Info_t and is shown in the following table:

Qp0l_Journal_Extended_Info_t			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	CHAR(1)	Journaling status
1	1	CHAR(1)	Options
2	2	CHAR(10)	Journal identifier (JID)
12	0C	CHAR(10)	Current or last journal name
22	16	CHAR(10)	Current or last journal library name
32	20	BINARY(4), UNSIGNED	Last journaling start time
36	24	CHAR(10)	Starting journal receiver for apply
46	2E	CHAR(10)	Starting journal receiver library name
56	38	CHAR(10)	Starting journal receiver ASP device
66	42	CHAR(1)	Apply journaled changes required
67	43	CHAR(1)	Rollback was ended
68	44	CHAR(12)	Reserved

Apply journaled changes required. Whether the object was restored with partial transactions which would require an Apply Journaled Changes (APYJRNCHG) command to complete the transaction. A partial transaction can occur if an object was saved using save-while-active requesting that transactions with pending record changes do not have to reach a commit boundary before the object is saved. The valid values are:

x'00' QP0L_APYJRNCHG_REQ_NO: The object does not have partial transactions.
x'01' QP0L_APYJRNCHG_REQ_YES: The object was restored with partial transactions. This object can not be used until the Apply Journaled Changes (APYJRNCHG) or Remove Journaled Changes (RMVJRNCHG) command is used to complete or rollback the partial transactions.

Current or last journal library name. If the value of the journaling status is QP0L_JOURNALED, then this field contains the name of the library containing the

currently used journal. If the value of the journaling status is QP0L_NOT_JOURNALED, then this field contains the name of the library containing the last used journal. All bytes in this field will be set to binary zero if this object has never been journaled.

Current or last journal name. If the value of the journaling status is QP0L_JOURNALED, then this field contains the name of the journal currently being used. If the value of the journaling status is QP0L_NOT_JOURNALED, then this field contains the name of the journal last used for this object. All bytes in this field will be set to binary zero if this object has never been journaled.

Journal identifier (JID). This field associates the object being journaled with an identifier that can be used on various journaling-related commands and APIs.

Journaling status. Current journaling status of the object. This field will be one of the following values:

x'00' QP0L_NOT_JOURNALED: The object is currently not being journaled.
x'01' QP0L_JOURNALED: The object is currently being journaled.

Last journaling start time. The number of seconds since the Epoch that corresponds to the last date and time for which the object had journaling started for it. This field will be set to binary zero if this object has never been journaled.

Options. This field describes the current journaling options. This field is composed of several bit flags and contains one or more of the following bit values:

x'80' QP0L_JOURNAL_SUBTREE: When this flag is returned, this object is a directory with IFS journaling subtree semantics. New objects created within this directory's subtree will inherit the journaling attributes and options from this directory.
x'08' QP0L_JOURNAL_OPTIONAL_ENTRIES: When journaling is active, entries that are considered optional are journaled. The list of optional journal entries varies for each object type. See the Integrated file system information in the Files and file systems topic for information regarding these optional entries for various objects.
x'20' QP0L_JOURNAL_AFTER_IMAGES: When journaling is active, the image of the object after a change is journaled.
x'40' QP0L_JOURNAL_BEFORE_IMAGES: When journaling is active, the image of the object prior to a change is journaled.

Reserved. A reserved field. This field will be set to binary zero.

Rollback was ended. Whether the object had rollback ended prior to completion of a request to roll back a transaction. The valid values are:

x'00' QP0L_ROLLBACK_END_NO: The object did not have a rollback operation ended prior to completion of a request to roll back a transaction.
x'01' QP0L_ROLLBACK_END_YES: The object had a rollback operation ended using the "End Rollback" option on the Work with Commitment Definition (WRKCMTDFN) screnn. It is recommended that the object be restored as it can not be used. As a last resort, the Change Journaled Object (CHGJRNOBJ) command can be used to allow the object to be used. Doing this, however, may leave the object in an inconsistent state.

Starting journal receiver ASP device. The name of the ASP for the library that contains the starting journal receiver. This field will be blank if no information is available. The valid values are:

**SYSBAS* The journal receiver library resides in the system or user ASPs
ASP device The journal receiver library resides in this ASP.

Starting journal receiver for apply. The oldest journal receiver needed to successfully Apply Journaled Changes (APYJRNCHG). When the Apply journaled Changes required field is set to QP0L_APYJRNCHG_REQ_YES the journal receiver contains the journal entries representing the start of the partial transaction. Otherwise; the journal receiver contains the journal entries representing the start-of-the-save operation. This field will be blank if no information is available.

Starting journal receiver library name. The name of the library that contains the journal receiver. This field will be blank if no information is available.

41 QP0L_ATTR_CRTOBJAUD: (CHAR(10)) The create object auditing value associated with the directory. This is the auditing value given to any objects created in the directory. Valid values are:

**SYSVAL* QP0L_AUD_SYSVAL: The object auditing value for the objects created in the directory is determined by the system auditing value (QCRTOBJAUD).

**NONE* QP0L_AUD_NONE: No auditing occurs for this object when it is read or changed regardless of the user who is accessing the object.

**USRPRF* QP0L_AUD_USRPRF: Audit this object only if the current user is being audited. The current user is tested to determine if auditing should be done for this object. The user profile can specify if only change access is audited or if both read and change accesses are audited for this object. The OBJAUD parameter of the Change User Auditing (CHGUSRAUD) command is used to change the auditing for a specific user.

**CHANGE* QP0L_AUD_CHANGE: Audit all change access to this object by all users on the system.

**ALL* QP0L_AUD_ALL: Audit all access to this object by all users on the system. All access is defined as a read or change operation.

**NOTAVL* QP0L_AUD_NOTAVL: The user performing the operation is not allowed to retrieve the current create object auditing value.

Note: The user must have all object (*ALLOBJ) or audit (*AUDIT) special authority to retrieve the create object auditing value.

42 QP0L_ATTR_SYSTEM_USE: (CHAR(1)) Whether the file has a special use by the system. This attribute is valid only for stream files. Possible values are:

x'00' QP0L_SYSUSE_NONE: The file is a generic stream file.

x'01' QP0L_SYSUSE_VRTVOL: The file is a virtual volume. Examples include tape and optical virtual volumes.

x'02' QP0L_SYSUSE_NWSSTG: The file is a network server storage space.

300 QP0L_ATTR_SUID: (CHAR(1)) Set effective user ID (UID) at execution time. This value is ignored if the specified object is a directory. Valid values are:

x'00' QP0L_SUID_OFF: The user ID (UID) is not set at execution time.

x'01' QP0L_SUID_ON: The object owner is the effective user ID (UID) at execution time.

301 QP0L_ATTR_SGID: (CHAR(1)) Set effective group ID (GID) at execution time. Valid values are:

x'00' QP0L_SGID_OFF: If the object is a file, the group ID (GID) is not set at execution time. If the object is a directory in the "root" (/), QOpenSys, and user-defined file systems, the group ID (GID) of objects created in the directory is set to the effective GID of the thread creating the object. This value cannot be set for other file systems.

x'01' QP0L_SGID_ON: If the object is a file, the group ID (GID) is set at execution time. If the object is a directory, the group ID (GID) of objects created in the directory is set to the GID of the parent directory.

Number of requested attributes. The total number of requested attributes that **Qp0lGetAttr()** returns. This field is required when the *Attr_Array_ptr* parameter is not NULL and must equal the number of constants in the array to which it points. When this field is zero, **Qp0lGetAttr()** returns all the attributes that are supported by the API and that are available for the object.

Buffer_ptr

(Input) A pointer to a buffer that the caller allocates for **Qp0lGetAttr()** to return the requested data. The caller also sets the *Buffer_Size_Provided* parameter to the number of bytes that are allocated for this buffer.

If the buffer provided is not large enough to hold all of the requested data, **Qp0lGetAttr()** fills the buffer with as much data as possible and sets the value pointed to by the *Buffer_Size_Needed_ptr* parameter equal to the number of bytes required for all of the requested data to be returned.

When the *Buffer_ptr* is NULL, **Qp0lGetAttr()** returns the total number of bytes needed to hold all of the requested attributes and sets the *Buffer_Size_Needed_ptr* parameter to point to this value.

Qp0lGetAttr() identifies each entry that it returns in the buffer with the constant that the user supplied in the input structure pointed to by the *Attr_Array_ptr* parameter. **Qp0lGetAttr()** returns this constant in the Attribute identification field. The constant must be used to identify the returned attribute because the attributes are returned in any order.

Qp0lGetAttr() fills the buffer with an entry for each requested attribute in the following format:

Buffer Pointer			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Offset to next attribute entry
4	4	BINARY(4)	Attribute identification
8	8	BINARY(4)	Size of attribute data
12	C	CHAR(4)	Reserved
16	10	CHAR(*)	Attribute data

Attribute data. The attribute data that was requested.

Attribute identification. The constant that identifies the returned attribute. Valid values follow and are the same constants as provided by the caller of **Qp0lGetAttr()**, pointed to by the *Attr_Array_ptr* parameter.

See the *Attr_Array_ptr* (page 326) parameter for descriptions of each of these attribute values.

0 QP0L_ATTR_OBJTYPE
1 QP0L_ATTR_DATA_SIZE
2 QP0L_ATTR_ALLOC_SIZE
3 QP0L_ATTR_EXTENDED_ATTR_SIZE
4 QP0L_ATTR_CREATE_TIME
5 QP0L_ATTR_ACCESS_TIME
6 QP0L_ATTR_CHANGE_TIME
7 QP0L_ATTR_MODIFY_TIME
8 QP0L_ATTR_STG_FREE
9 QP0L_ATTR_CHECKED_OUT

10	QP0L_ATTR_LOCAL_REMOTE
11	QP0L_ATTR_AUTH
12	QP0L_ATTR_FILE_ID
13	QP0L_ATTR_ASP
14	QP0L_ATTR_DATA_SIZE_64
15	QP0L_ATTR_ALLOC_SIZE_64
16	QP0L_ATTR_USAGE_INFORMATION
17	QP0L_ATTR_PC_READ_ONLY
18	QP0L_ATTR_PC_HIDDEN
19	QP0L_ATTR_PC_SYSTEM
20	QP0L_ATTR_PC_ARCHIVE
21	QP0L_ATTR_SYSTEM_ARCHIVE
22	QP0L_ATTR_CODEPAGE
23	QP0L_ATTR_FILE_FORMAT
24	QP0L_ATTR_UDFS_DEFAULT_FORMAT
25	QP0L_ATTR_JOURNAL_INFORMATION
26	QP0L_ATTR_ALWCKPWRT
27	QP0L_ATTR_CCSID
28	QP0L_ATTR_SIGNED
29	QP0L_ATTR_SYS_SIGNED
30	QP0L_ATTR_MULT_SIGS
31	QP0L_ATTR_DISK_STG_OPT
32	QP0L_ATTR_MAIN_STG_OPT
33	QP0L_ATTR_DIR_FORMAT
34	QP0L_ATTR_AUDIT
35	QP0L_ATTR_CRTOBJSCAN
36	QP0L_ATTR_SCAN
37	QP0L_ATTR_SCAN_INFO
38	QP0L_ATTR_ALWSAV
39	QP0L_ATTR_RSTDRNMUNL
40	QP0L_ATTR_JOURNAL_EXTENDED_INFORMATION
» *41*	QP0L_ATTR_CRTOBJAUD «
» *42*	QP0L_ATTR_SYSTEM_USE «
300	QP0L_ATTR_SUID
301	QP0L_ATTR_SGID

Offset to next attribute entry. The offset to the next attribute entry in the buffer. This offset is relative to the start of the buffer. An offset of zero means that no more attribute entries follow.

Reserved. A reserved field set to binary zero.

Size of attribute data. The total size of all the data for this attribute. The special value of 0 in this field indicates that the attribute is not supported by the file system in which the object is stored. The attribute data is padded with hexadecimal zeros. The size indicated in this field does not include the padding bytes.

Buffer_Size_Provided

(Input) The number of bytes the caller allocates in a buffer for the return of requested data. The buffer is pointed to by the *Buffer_ptr* parameter.

If this size is set to zero or is not large enough to hold all of the requested data, **Qp0lGetAttr()** fills the buffer with as much data as possible and sets the value pointed to by the *Buffer_Size_Needed_ptr* parameter equal to the number of bytes required for all of the requested data to be returned.

When determining the appropriate allocation, the caller should assume that the returned attribute data will be aligned on a minimum of an 8-byte boundary.

Buffer_Size_Needed_ptr
(Output) A pointer to the number of bytes that the caller needs to allocate for **Qp0lGetAttr()** to return all of the requested data.

Num_Bytes_Returned_ptr
(Output) A pointer to the actual number of bytes of data returned in the user buffer. This field is zero if the *Buffer_ptr* parameter is NULL.

Follow_Symlnk
(Input) If the last component in the *Path_Name* is a symbolic link, this parameter determines if the symbolic link or the path contained in the symbolic link is acted upon: Valid values are:

0 QP0L_DONOT_FOLLOW_SYMLNK: A symbolic link in the last component is not followed. Attributes of the symbolic link object are returned.

1 QP0L_FOLLOW_SYMLNK: A symbolic link in the last component is followed. The attributes of the object contained in the symbolic link are returned.

Authorities

Note: Adopted authority is not used.

Authorization Required for Qp0lGetAttr()		
Object Referred to	**Authority Required**	*errno*
Each directory, preceding the last component, in the *Path_Name* » (except when only QP0L_ATTR_AUDIT and/or QP0L_ATTR_CRTOBJAUD are requested) «	*X	EACCES
Object, when retrieving the QP0L_ATTR_AUTH attribute	*OBJMGT	EACCES
Note: If the file system supports *ALLOBJ special authority and if you have *ALLOBJ special authority, you do not need the listed object authority.		

Return Value

0 **Qp0lGetAttr()** was successful.

-1 **Qp0lGetAttr()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **Qp0lGetAttr()** is not successful, *errno* indicates one of the following errors:

Error condition	**Additional information**
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECANCEL (page 543)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	

Error condition	Additional information
[EFAULT (page 541)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EOFFLINE (page 545)]	
[EOVERFLOW (page 546)]	
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[EUNKNOWN (page 544)]	

Additionally, if interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. ″Root″ (/), QOpenSys, and User-Defined File System Differences

   The QP0L_ATTR_ALLOC_SIZE and QP0L_ATTR_ALLOC_SIZE_64 values can be influenced by the setting of the disk storage option attribute. See QP0L_ATTR_DISK_STG_OPT for more information.
3. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   Qp0lGetAttr() could return zero for the QP0L_ATTR_ACCESS_TIME value (in the buffer area) under some conditions.

   Refer to the CL Programming topic for more information regarding which object types maintain usage information that is returned for the QP0L_ATTR_USAGE_INFORMATION attribute.

   When **Qp0lGetAttr()** is performed on a physical file member, the QP0L_ATTR_JOURNAL_INFORMATION or QP0L_ATTR_JOURNAL_EXTEND_INFORMATION attribute will contain journaling information applicable to the physical file that contains the member.
4. QFileSvr.400 File System Differences

   If only the QP0L_ATTR_AUDIT or QP0L_ATTR_CRTOBJAUD attributes are requested, the QSECOFR user profiles on the source and target system must be enabled, and their passwords must match for the operation to succeed.
5. Network File System Differences

   If the user has the appropriate authority when requesting the QP0L_ATTR_AUDIT attribute for objects in the Network File System, the value QP0L_AUD_NONE will always be returned.

Related Information

- The <**Qp0lstdi.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**qlg.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "Integrated File System Scan on Close Exit Program" on page 513
- "Integrated File System Scan on Open Exit Program" on page 523.
- "fstat()—Get File Information by Descriptor" on page 95—Get File Information by Descriptor
- "lstat()—Get File or Link Information" on page 162—Get File or Link Information
- "QlgGetAttr()—Get Attributes (using NLS-enabled path name)" on page 247—Get Attributes (using NLS-enabled path name)
- "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293—Get File Information (using NLS-enabled path name)

- "QlgLstat()—Get File or Link Information (using NLS-enabled path name)" on page 265—Get File or Link Information (using NLS-enabled path name)
- "Qp0lSetAttr()—Set Attributes" on page 403—Set Attributes
- Retrieve System Values (QWCRSVAL) API
- "stat()—Get File Information" on page 468—Get File Information

Example

See Code disclaimer information for information pertaining to code examples.

Following is an example showing a call to **Qp0lGetAttr()**. The example also shows a call to **Qp0lSaveStgFree()**.

```
/*****************************************************************/
#include "Qp0lstdi.h"
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/types.h>
#include <qusec.h>
#include <time.h>

int Save(Qp0l_Pathnames_t *Path_name_ptr)
{
  /***************************************************************/
  /* No function here in the example                             */
  /***************************************************************/
};

void SaveAnObject(Qp0l_Pathnames_t *Path_name_ptr,
                  int              *Return_code_ptr,
                  int              *Return_value_ptr,
                  void             *Function_CtlBlk_ptr)
{
  /***************************************************************/
  /* This function saves a file and its hard links to tape.     */
  /***************************************************************/
  int rc;

  if ((Path_name_ptr == (Qp0l_Pathnames_t *)NULL) ||
      (Path_name_ptr->Number_Of_Names == 0))
  {
    printf("In User Exit Program with null Path \n");
  }
  else
  {
    /* This example calls a function (Save) that could call the  */
    /* Save Object (QsrSave) API.  The QsrSave API is designed to */
    /* save a copy of one or more objects that can be used in the */
    /* integrated file system.  For details on using QsrSave, see  */
    /* the Backup and Recovery API part.                          */

    rc = (Save(Path_name_ptr));

    *Return_code_ptr = rc;
    *Return_value_ptr = errno;

    if (rc == 0)
    {
      /* Other processing for a successfully saved object. */
    }
    else
    {
      /* Optional processing such as storing information   */
      /* to be returned to the caller in the function      */
```

```
      /* control block area, or building a list of the     */
      /* files whose save attempts failed, or other.       */
    }
  }
  return;
}                               /* end SaveAnObject exit program */

int main (int argc, char *argv[])
{
#define MYPN "ADIR/ASTMF"
  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  const char Path_Name_Del_const[2] = "/";

  struct pnstruct
  {
    Qlg_Path_Name_T  qlg_struct;
    char             pn[1];
  };
  struct pnstruct pns;
  struct pnstruct *pns_ptr = NULL;

  struct attrStruct
  {
    Qp0l_AttrTypes_List_t  attr_struct;
    uint AttrTypes[10];
  };
  struct attrStruct Attr_types_ptr;
  Qp0l_Attr_Header_t *attrPtr;
  char *attrValp;

  Qp0l_StgFree_Function_t  User_function;

  struct
  {
    uint    AnyData_to_the_exitprogram;
    uint    AnyData_not_processed_by_the_API;
  } CtlBlkAreaName;

  time_t mytime;
  char BufferArea[250];
  unsigned int buff_size_provided;
  unsigned int buff_size_needed = 0;
  unsigned int num_bytes_returned = 0;
  unsigned int follow_sym;
  int done=0;
  int rc;
  int returned_data_index = 0;

  /*************************************************************/
  /* Initialize Get Attributes Parameters                      */
  /*************************************************************/
  memset((void*)&pns, 0x00, sizeof(struct pnstruct));
  pns.qlg_struct.CCSID = 37;
  memcpy(pns.qlg_struct.Country_ID,US_const,2);
  memcpy(pns.qlg_struct.Language_ID,Language_const,3);
  pns.qlg_struct.Path_Type = 0;
  pns.qlg_struct.Path_Length = sizeof(MYPN)-1;
  memcpy(pns.qlg_struct.Path_Name_Delimiter,Path_Name_Del_const,1);
  memcpy(pns.pn,MYPN,sizeof(MYPN));
  memset((void *)&Attr_types_ptr, 0x00,sizeof(struct attrStruct));
  pns_ptr = &pns;

  Attr_types_ptr.attr_struct.Number_Of_ReqAttrs = 2;
  Attr_types_ptr.AttrTypes[0] = QP0L_ATTR_ACCESS_TIME;
  Attr_types_ptr.AttrTypes[1] = QP0L_ATTR_STG_FREE;
```

```
buff_size_provided = 250;

follow_sym = QP0L_FOLLOW_SYMLNK;

/**************************************************************/
/* Call the Qp0lGetAttr() API to retrieve attributes to       */
/* determine if selection criteria can be met for calling     */
/* the Qp0lSaveStgFree() API.                                 */
/**************************************************************/
rc = Qp0lGetAttr((Qlg_Path_Name_T *)&pns,
                 (Qp0l_AttrTypes_List_t *)&Attr_types_ptr,
                 BufferArea,
                 buff_size_provided,
                 &buff_size_needed,
                 &num_bytes_returned,
                 follow_sym);
if (rc == 0)                      /* check API return code */
{
  /* Must first check if any data was returned. */
  if (num_bytes_returned > 0)
  {
    attrPtr = (Qp0l_Attr_Header_t *)BufferArea;
    while(!done)
    {
      attrValp = (char *)attrPtr +
        sizeof(Qp0l_Attr_Header_t);  /* Point to attr value   */
      /****************************************************/
      /* The following code prints the two attributes that  */
      /* were returned.  Add more code here, for example,   */
      /* to determine if the returned attributes meet       */
      /* the criteria or policies for storage freeing.      */
      /****************************************************/
      printf ("****************************************\n");
      printf ("Attr ID #%d = %d - ",
              returned_data_index,
              attrPtr->Attr_ID);
      if(attrPtr->Attr_Size > 0)
      {
        switch (attrPtr->Attr_ID)
        {
          case QP0L_ATTR_ACCESS_TIME:
              printf("QP0L_ATTR_ACCESS_TIME\n");
              memcpy((void *)&mytime,
                     (void *)attrValp,
                     attrPtr->Attr_Size);
              printf ("%s", ctime(&mytime));
              break;
          case QP0L_ATTR_STG_FREE:
              printf ("QP0L_ATTR_STG_FREE\n");
              switch (attrValp[0])
              {
                case QP0L_SYS_STG_FREE:
                    printf ("--Is storage freed--\n");
                    break;
                case QP0L_SYS_NOT_STG_FREE:
                    printf ("--Is not storage freed--\n");
                    break;
                default:
                    printf ("Invalid data: %d.\n",
                            attrValp[0]);
                    break;
              }
              break;
          default:
              printf ("Undefined return type (attr id unknown.)\n");
              break;
        }                      /* end switch                   */
```

```
        }
        else
          printf("Attribute has no value\n");
        printf("***Size of this attr's data: %d\n",
               attrPtr->Attr_Size);
        printf("***Offset to next attr: %d\n",
               attrPtr->Next_Attr_Offset);
        ++returned_data_index;
        if(attrPtr->Next_Attr_Offset > 0) /* If more data        */
          attrPtr = (Qp0l_Attr_Header_t *) /* Set attribute      */
            &(BufferArea[attrPtr->Next_Attr_Offset]); /* pointer */
        else                    /* No more data                  */
          done = 1;             /* End the loop                  */
      }

    /*********************************************************/
    /* Initialize Save Storage Free Parameters.  The path     */
    /* name parameter was already initialized as part of the  */
    /* call to Qp0lGetAttr() API and is assumed, in this      */
    /* example, to be the same pathname.  Both APIs require   */
    /* the same path name format.                             */
    /*********************************************************/
      memset((void *)&User_function,0x00,sizeof(Qp0l_StgFree_Function_t));
      User_function.Mltthdacn[0] = QP0L_MLTTHDACN_NOMSG;
      User_function.Function_Type = QP0L_USER_FUNCTION_PTR;
      User_function.Procedure = &SaveAnObject;

      rc = Qp0lSaveStgFree((Qlg_Path_Name_T *)&pns,
                           &User_function,
                           &CtlBlkAreaName);
      if(rc == 0)
        printf("Qp0lSaveStgFree() Successful!");
      else
      {/* Unsuccessful return from Qp0lSaveStgFree() API.  */
      /* The following code prints the errno value message.  */
        rc = errno;
        printf("ERROR on Qp0lSaveStgFree(): error = %d\n", rc);
        perror("Error message");
      }
    }                           /* if (num_bytes_returned > 0)   */
    else
      rc = EUNKNOWN;
  }                 /* end rcGA == 0, Qp0lGetAttr() was successful */
  else
  {
    rc = errno;
    printf("ERROR on Qp0lGetAttr(): error = %d\n", rc);
    perror("Error message");
  }
  return(rc);
} /* end main */
```

API introduced: V4R3

Top | UNIX-Type APIs | APIs by category

Qp0lGetPathFromFileID()—Get Path Name of Object from Its File ID

Syntax

```
#include <Qp0lstdi.h>

char *Qp0lGetPathFromFileID(char *buf, size_t size,
                            Qp0lFID_t fileid);
```

Service Program Name: QP0LLIB2
Default Public Authority: *USE
Threadsafe: Yes

The **Qp0lGetPathFromFileID()** function determines an absolute path name of the file identified by *fileid* and stores it in *buf*. The components of the returned path name are not symbolic links. If the file has more than one path name, only one is returned.

The access time of each directory in the absolute path name of the file (excluding the file itself) is updated.

If *buf* is a NULL pointer, **Qp0lGetPathFromFileID()** returns a NULL pointer and the EINVAL error.

The contents of *buf* after an error are not defined.

Qp0lGetPathFromFileID() is supported in the "root" (/), QOpenSys, and user-defined file systems.

Parameters

buf (Output) A pointer to a buffer that will be used to hold an absolute path name of the file identified by *fileid*. The buffer must be large enough to contain the full path name including the terminating NULL character.

The path name is returned in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgGetPathFromFileID()—Get Path Name of Object from Its File ID (using NLS-enabled path name)" on page 249 for a description and an example of supplying the *buf* in any CCSID.

size (Input) The number of bytes in the buffer `buf`.

fileid (Input) The identifier of the file whose path name is to be returned. This identifier is logged in audit journal entries to identify the file being audited. See the `Parent File ID` and `Object File ID` fields of the audit journal entries described in the iSeries Security Reference book.

Authorities

Note: Adopted authority is not used.

Authorization required for Qp0lGetPathFromFileID()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the file	*RX	EACCES
The file itself	*R	EACCES

Return Value

value **Qp0lGetPathFromFileID()** was successful. The value returned is a pointer to *buf*.

NULL **Qp0lGetPathFromFileID()** was not successful. The *errno* global variable is set to indicate the error. After an error, the contents of *buf* are not defined.

Error Conditions

If Qp0lGetPathFromFileID() is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ENOENT (page 540)]	No path names were found for this *fileid* or the user is not authorized to any of the paths.
[ENOMEM (page 543)]	
[ENOTAVAIL (page 547)]	
[ERANGE (page 540)]	For example, the **size** argument is too small. It is greater than zero but smaller than the length of the path name plus a NULL character.
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. The following file systems do not support **Qp0lGetPathFromFileID()**:

   - Network File System
   - QSYS.LIB
   - Independent ASP QSYS.LIB
   - QDLS
   - QOPT
   - QFileSvr.400
   - QNetWare
   - QNTC

Related Information

- The <**Qp0lstdi.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "QlgGetPathFromFileID()—Get Path Name of Object from Its File ID (using NLS-enabled path name)" on page 249—Get Path Name of Object from Its File ID (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines the path name of a file, given its file ID. In this example, the fileid is hardcoded. More realistically, the fileid is obtained from the audit journal entry and passed to **Qp0lGetPathFromFileID()**.

```
#include <Qp0lstdi.h>
#include <stdio.h>

main()
{
  char      path[1024];
  Qp0lFID_t  fileid = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                       0x00, 0x00, 0x00, 0x00, 0x80, 0xFF, 0xCF, 0x00};

  if (Qp0lGetPathFromFileID(path, sizeof(path), fileid) == NULL)
    perror("Qp0lGetPathFromFileID() error");
  else
    printf("The file's path is: %s\n", path);
}
```

Output:

```
The file's path is: /myfile
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

Qp0lOpen()—Open File

Syntax

```
#include <Qp0lstdi.h>

int Qp0lOpen(Qlg_Path_Name_T *Path_Name,
int oflag, . . .);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Theadsafe: Conditional; see Usage Notes for "open()—Open File" on page 195 API.

The **Qp0lOpen()** function, similar to the **open()** function, opens a file and returns a number called a **file descriptor**. **Qp0lOpen()** differs from **open()** in that the *Path_Name* parameter is a pointer to a Qlg_Path_Name_T structure instead of a pointer to a character string.

Only the *Path_Name* parameter is described here. For a discussion of the other parameters, authorities required, return values, and related information, see "open()—Open File" on page 195.

Note:: To use this API with large file APIs, you must specify the O_LARGEFILE flag on the *oflag* parameter.

Parameters

Path_Name
: (Input) The path name of the file to be opened. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path Name Format.

Related Information

- The <**fcntl.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "open()—Open File" on page 195—Open File
- "close()—Close File or Socket Descriptor" on page 34—Close File or Socket Descriptor

Example

See Code disclaimer information for information pertaining to code examples.

The following example creates and opens an output file for exclusive access. This program was stored in a source file with CCSID 37, so the constant string ″newfile″ will be compiled in coded character set identifier (CCSID) 37. Therefore, the country or region and language specified are United States English, and the CCSID specified is 37.

```
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>

main()
{
  int fildes;

  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  const char Path_Name_Del_const[2] = "/";

  struct pnstruct
    {
     Qlg_Path_Name_T  qlg_struct;
     char             pn[7];
    };
  struct pnstruct pns;
  struct pnstruct *pns_ptr = NULL;

  char fn[]="newfile";

  memset((void*)&pns, 0x00, sizeof(struct pnstruct));
  pns.qlg_struct.CCSID = 37;
  memcpy(pns.qlg_struct.Country_ID,US_const,2);
  memcpy(pns.qlg_struct.Language_ID,Language_const,3);;
  pns.qlg_struct.Path_Type = 0;
  pns.qlg_struct.Path_Length = sizeof(fn) - 1;
  memcpy(pns.qlg_struct.Path_Name_Delimiter,
         Path_Name_Del_const,1);
  memcpy(pns.pn,fn,sizeof(fn));

  pns_ptr = &pns;
  if(fildes = Qp0lOpen((Qlg_Path_Name_T *)pns_ptr,
           O_WRONLY|O_CREAT|O_EXCL, S_IRWXU)) == -1)
  {
        perror("Qp0lOpen() error");
  }

}
```

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

Qp0lProcessSubtree()—Process a Path Name

Syntax

```
#include <Qp0lstdi.h>

int Qp0lProcessSubtree (
 Qlg_Path_Name_T         *Path_Name,
 uint                     Subtree_level,
 Qp0l_Objtypes_List_t    *Objtypes_array_ptr,
 uint                     Local_remote_obj,
 Qp0l_IN_EXclusion_List_t *IN_EXclusion_ptr,
 uint                     Err_recovery_action,
 Qp0l_User_Function_t    *UserFunction_ptr,
 void                    *Function_CtlBlk_ptr, ...);
```

Service Program Name: QP0LLIB2
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 362.

The **Qp0lProcessSubtree()** function searches the directory tree under a specific path name. It selects and passes objects, one at a time, to an exit program that is identified on its call. The exit program can be either a procedure or a program.

Qp0lProcessSubtree() performs recursive read operations to access any object in any file system. The order in which objects are selected and passed to the exit program can vary within a given file system and within a given directory, dependent on file system rules. The only guaranteed ordering is that all selected objects within a given directory are passed to the exit program before the parent directory is passed to the exit program.

Parameters

Path_Name
(Input) The path name where **Qp0lProcessSubtree()** starts its search. All relative path names are relative to the current directory at the time of the call to **Qp0lProcessSubtree()**. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path Name Format. The *Path_Name* parameter must be NULL to use the *IN_EXclusion_ptr* parameter to enter multiple path names for inclusion on a single call to **Qp0lProcessSubtree()**.

Subtree_level
(Input) An unsigned integer that tells **Qp0lProcessSubtree()** whether or not to open subdirectories in the path being processed. Valid values follow:

0 **QP0L_SUBTREE_YES:** All subdirectories are opened by **Qp0lProcessSubtree()** so that the objects they contain are sent to the exit program if they meet the caller's selection criteria.

1 **QP0L_SUBTREE_NO:** Only first-level objects are processed. The names of subdirectories, which meet the selection criteria, are passed to the exit program, but they are not opened by **Qp0lProcessSubtree()**. Thus, the objects the subdirectories contain are not matched against selection criteria and therefore are not sent to the exit program.

Objtypes_array_ptr
(Input) A pointer to an array of object types. Each entry in the array identifies an object type that **Qp0lProcessSubtree()** uses to determine what will be passed to the exit program. The Number of

object types field contains the total number of object types in the array. A NULL pointer means that there is no filtering according to object type and that all object types that meet other selection criteria are passed to the exit program.

The structure for this parameter follows.

Object Types Array Pointer

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Number of object types
4	4	ARRAY(*) of CHAR(11)	Array of object types structure

Array of object types structure
An array identifying each object type used to determine what will be passed to the exit program when processing a path. Each entry is limited to 11 characters, including a NULL terminator, and is padded with blanks. Object types must be entered in standard object type format which is all capital letters, preceded by an asterisk (*). For a complete list of the available object types, see Object Types in the CL topic.

Qp0lProcessSubtree() verifies that valid object types are entered and returns the *errno* EINVAL when an object type that is not valid is entered. Although some object types are scoped to a specific file system, **Qp0lProcessSubtree()** does not validate object types according to file systems.

Valid special values for this parameter follow:

***ALLDIR:** Select all directory object types. This includes *LIB, *DIR, *FLR, *FILE, and *DDIR object types.

***ALLQSYS:** Select all QSYS.LIB object types. This includes all objects in the QSYS.LIB file system and all independent ASP QSYS.LIB file systems which are available when the API is first called.

Note: *IN_EXclusion_ptr* must also be specified as an inclusion array. If *NOQSYS is specified, *ALLQSYS cannot also be specified.

***ALLSTMF:** Select all stream file object types. This includes *MBR, *DOC, *STMF, *DSTMF, and *USRSPC object types.

***MBR:** Select all database file member types.

***NOQSYS:** Exclude all QSYS.LIB object types. This includes all objects in the QSYS.LIB file system and all independent ASP QSYS.LIB file systems which are available when the API is first called.

Note: This special value only has meaning if '/' or '/asp_name' is specified for the *Path_Name* parameter (where asp_name is the name of an independent ASP which is available when the API is first called). Additionally, if *IN_EXclusion_ptr* is specified, it must only be as an exclusion array. If *ALLQSYS is specified, *NOQSYS cannot also be specified.

Number of object types
The number of types included in the search.

Local_remote_obj
(Input) An unsigned integer that tells **Qp0lProcessSubtree()** whether to select only local objects, only remote objects, or both. Note that the decision of whether a file is local or remote varies according to the respective file system rules. Objects in file systems that do not carry either a local or remote indicator are treated as remote. Valid values follow:

0 **QP0L_LOCAL_REMOTE_OBJ:** Both local and remote objects are passed to the exit program.

1 **QP0L_LOCAL_OBJ:** Only local objects are passed to the exit program.
2 **QP0L_REMOTE_OBJ:** Only remote objects are passed to the exit program.

IN_EXclusion_ptr
(Input) A pointer to an array of pointers. Each pointer in the array points to a specific path name that identifies a directory, and all of its subdirectories, that **Qp0lProcessSubtree()** either includes or excludes in its search to find objects that meet the caller's input criteria. If this pointer is not NULL, the IN_EXclusion pointer type must indicate whether the list is an inclusive or exclusive list. The Number of pointers field must contain the number of path names for inclusion or exclusion on the search.

Use an inclusive list to specify multiple path names for searches on a single call to **Qp0lProcessSubtree()** versus using the *Path_Name* parameter, which searches only one path per call. The *Path_Name* parameter and an inclusive list are mutually exclusive. EINVAL is returned if both parameters are specified. The *IN_EXclusion_ptr* must be NULL if not used. All of the rules that apply to a single *Path_Name* entry apply to each inclusive list entry.

While an inclusion list allows the caller of **Qp0lProcessSubtree()** to identify multiple path names for processing, **Qp0lProcessSubtree()** does not perform any verification to ensure uniqueness of path names or to verify any other relationship between path names entered in the inclusion array. For example, if the path names entered represent nested directories, **Qp0lProcessSubtree()** calls the exit program multiple times without any error message or other notification of this nesting.

Specify the root directory for a given file system as an exclusive list entry to eliminate that file system from a search.

All relative path names are relative to the current directory of the job that calls **Qp0lProcessSubtree()**.

The structure for this parameter follows.

IN_EXclusion Pointer

This points to a list of path names to either include or exclude from a search.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	IN_EXclusion pointer type
4	4	BINARY(4)	Number of pointers
8	8	CHAR(8)	Reserved
16	10	ARRAY(*)	Path name pointers

IN_EXclusion pointer type
Whether a path name array contains directories that are included or contains directories that are excluded. Valid values follow:

0 **QP0L_INCLUSION_TYPE:** An inclusion array is identified.
1 **QP0L_EXCLUSION_TYPE:** An exclusion array is identified.

Number of pointers
The number of path name pointers that are in the inclusion or exclusion array.

Path name pointers
An array of pointers. Each pointer points to a path name that is included or excluded. Each path name must follow the Qlg_Path_Name_T structure. For more information on this structure, see Path Name Format.

Reserved
A reserved field. This field must be set to binary zero.

Err_recovery_action
(Input) An unsigned integer that describes how **Qp0lProcessSubtree()** handles errors that are not severe enough to force the API to end processing. Valid values follow:

0 **QP0L_PASS_WITH_ERRORID:** Calls the exit program and specifies the name (when the name is available) of the object being accessed when an error occurs. This value also sends a valid *errno* to the exit program.

1 **QP0L_BYPASS_NO_ERRORID:** Bypasses the object being accessed when an error occurs, and moves to process the next object in the tree without notification to the calling program or to the exit program that an error has occurred.

2 **QP0L_JOBLOG_NO_ERRORID:** Sends message CPDA1C0 to the job log to identify the object being accessed when an error occurs. This value returns to process the next object without notification to the calling program or to the exit program that an error has occurred.

3 **QP0L_NULLNAME_ERRORID:** Calls the exit program with a NULL object name and a valid *errno*.

4 **QP0L_END_PROCESS_SUBTREE:** Quits **Qp0lProcessSubtree()** when an error occurs, and returns to the calling program, regardless of the error type. Note that the exit program is still given a call but cannot override the caller's decision to end processing. Calling the exit program allows the exit program to perform other tasks before the API returns to the caller. For example, the exit program can put information in the function control block that can be processed by the caller when the caller regains control.

UserFunction_ptr
(Input) A pointer to the name of an exit program that the caller wants **Qp0lProcessSubtree()** to call upon finding an object that matches the selection criteria. This exit program can be either a procedure or a program. See "Process a Path Name Exit Program" on page 533 for the syntax of the user exit program.

The structure for this parameter follows.

User Function Pointer

This points to the user exit program. The exit program can be a procedure or a program.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Function type flag
4	4	CHAR(10)	Program library
14	E	CHAR(10)	Program name
24	18	CHAR(1)	Multithreaded job action
25	19	CHAR(7)	Reserved
32	20	PP(*)	Procedure pointer to the exit procedure

Function type flag
An unsigned integer that indicates whether the user-supplied exit program that is called by **Qp0lProcessSubtree()** is a procedure or a program. Valid values follow:

0 **QP0L_USER_FUNCTION_PTR:** A user procedure is called.

1 **QP0L_USER_FUNCTION_PGM:** A user program is called.

Multithreaded job action
(Input) A CHAR(1) value that indicates the action to take in a multithreaded job. The default value is QP0L_MLTTHDACN_SYSVAL. For release compatibility and for

processing this parameter against the QMLTTHDACN system value, x'00, x'01', x'02', & x'03' are treated as x'F0', x'F1', x'F2', and x'F3'. Valid values follow:

x'00' QP0L_MLTTHDACN_SYSVAL: The API evaluates the QMLTTHDACN system value to determine the action to take in a multithreaded job. Although the API can make repetitive calls to an exit program, the system value is evaluated once before Qp0lProcessSubtree() issues its first exit program call. This value is used on subsequent calls until the API returns control to its caller. Valid QMLTTHDACN system values follow:

- *'1'* Call the exit program. Do not send an informational message.
- *'2'* Call the exit program. Send informational message CPI3C80. Qp0lProcessSubtree() may call the exit program multiple times; however, this message is sent only once for each call to Qp0lProcessSubtree().
- *'3'* The exit program is not called when the API determines that it is running in a multithreaded job. ENOTSAFE is returned.

x'01' QP0L_MLTTHDACN_NOMSG: Call the exit program. Do not send an informational message.

x'02' QP0L_MLTTHDACN_MSG: Call the exit program. Send informational message CPI3C80. Qp0lProcessSubtree() may call the exit program multiple times; however, this message is sent only once for each call to Qp0lProcessSubtree().

x'03' QP0L_MLTTHDACN_NO: The exit program is not called when the API determines that it is running in a multithreaded job. ENOTSAFE is returned.

Procedure pointer to the exit procedure
A procedure pointer to the procedure that **Qp0lProcessSubtree()** calls. This field must be NULL if a program is called instead of a procedure.

Program library
The library in which the called program, identified by Program name, is located. This field must be blank if a procedure is called instead of a program.

Program name
The name of the program that is called. The program is located in the library identified by Program library. This field must be blank if a procedure is called instead of a program.

Reserved
A reserved field. This field must be set to binary zero.

Function_CtlBlk_ptr
(Input) A pointer that **Qp0lProcessSubtree()** passes to the user-defined exit program that is called. **Qp0lProcessSubtree()** does not process this pointer or what is referred to by the pointer. It passes the pointer as a parameter to the user-defined exit program that was specified. This is a means for the caller of **Qp0lProcessSubtree()** to pass information to and from the Process a Path Name exit program.

Authorities

Note: Adopted authority is not used.

Authorization Required for Qp0lProcessSubtree()

Object Referred to	Authority Required	errno
Each directory, preceding the last component, in a *Path Name*	*X	EACCES
The *Path Name* directory and all subdirectories of the *Path Name* that are included in the search.	*RX (See **Note**)	EACCES
Each directory, preceding the last component, in any path name pointed to by the *IN_EXclusion ptr*	*X	EACCES

Object Referred to	**Authority Required**	**errno**
The *Path Name* directory and all subdirectories of any path name pointed to by an inclusive list	*RX (See **Note**)	EACCES
The object identified by the path name that is passed to the exit program, if the object is a user profile (*USRPRF)	Any authority greater than *EXCLUDE	EACCES
Any called program pointed to by the *UserFunction_ptr* parameter	*X	EACCES
Any library that contains the called program pointed to by the *UserFunction_ptr* parameter	*X	EACCES
Note: If the directory or subdirectories have no objects in them, only *R is required.		

Return Value

0 **Qp0lProcessSubtree()** was successful.

-1 **Qp0lProcessSubtree()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **Qp0lProcessSubtree()** is not successful, the *errno* indicates one of the following errors:

Error condition	**Additional information**
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EISDIR (page 544)]	
[ELOOP (page 544)]	
[EMFILE (page 543)]	
[ENAMETOOLONG (page 544)]	
[ENFILE (page 543)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOSYSRSC (page 545)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[EUNKNOWN (page 544)]	

Error Messages

The following message may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPFA0D4 E	File system error occurred. Error number &1.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. If the exit program called by **Qp0lProcessSubtree()** is not threadsafe or uses a function that is not threadsafe, then **Qp0lProcessSubtree()** is not threadsafe.
3. If the exit program called by **Qp0lProcessSubtree()** uses a function that fails when there are secondary threads active in the job, **Qp0lProcessSubtree()** may fail as a result.
4. Basic function and usage considerations
   - **Qp0lProcessSubtree()** does not perform the following tasks but is designed to work with the user exit function and other APIs to be useful in accomplishing the following and other tasks:
     - Retrieve object attributes (like authorities, dates, or sizes).
     - Build lists from selected objects.
     - Delete directories.
     - Identify multiple occurrences of an object within or across directories.
     - Count the number of objects in a directory.
   - **DosSetRelMaxFH()** is called to increase to the maximum the number of file descriptors that can be opened during processing such that **Qp0lProcessSubtree()** is not likely to fail due to a lack of descriptors. This value is not reset when **Qp0lProcessSubtree()** ends because the API could be running in a multithreaded job.
5. Object locking

   Qp0lProcessSubtree() does not perform any object locking, other than what is done when opening a directory to read the objects it contains, so that the exit program does not encounter or need to manage locks held by **Qp0lProcessSubtree()**.

   If **Qp0lProcessSubtree()** encounters a directory that is locked, **Qp0lProcessSubtree()** uses the defined Err_recovery_action to determine how to handle the locked condition. Locks on objects that are not directories have no effect on **Qp0lProcessSubtree()**.
6. Search Results

Once **Qp0lProcessSubtree()** has started searching a path, its search results may be affected by operations that update the organization of objects within the specifed directory tree. This includes but is not limited to the following:

- Adding, removing, or renaming object links,
- Mounting or unmounting file systems,
- Updating the effective root directory for the process calling this API,
- Updating the contents of a symbolic link.

7. Design considerations for parameters

- Symbolic links

  When the last component of the path name supplied on the initial call of **Qp0lProcessSubtree()** is a symbolic link, **Qp0lProcessSubtree()** resolves and follows the initial link to its target and performs its normal functions on the target. All other symbolic links that are encountered in the same search are not resolved to their targets.

  If the path name supplied on the initial call of **Qp0lProcessSubtree()** is a symbolic link that points to another file system or that points to a remote file system, the API resolves and processes the initial link only. It does not resolve other symbolic links that are encountered in the same search. However, if the caller specified that remote objects are not processed, but the initial path name (whether a symbolic link or not) points to a remote file system, the link is not resolved. **Qp0lProcessSubtree()** calls the exit program with a NULL path name and an indicator that **Qp0lProcessSubtree()** has completed successfully without any error indicators to the exit program.

  When *SYMLNK is specified as part of the selection criteria, **Qp0lProcessSubtree()** does not resolve the selected names.

- Recovery Actions

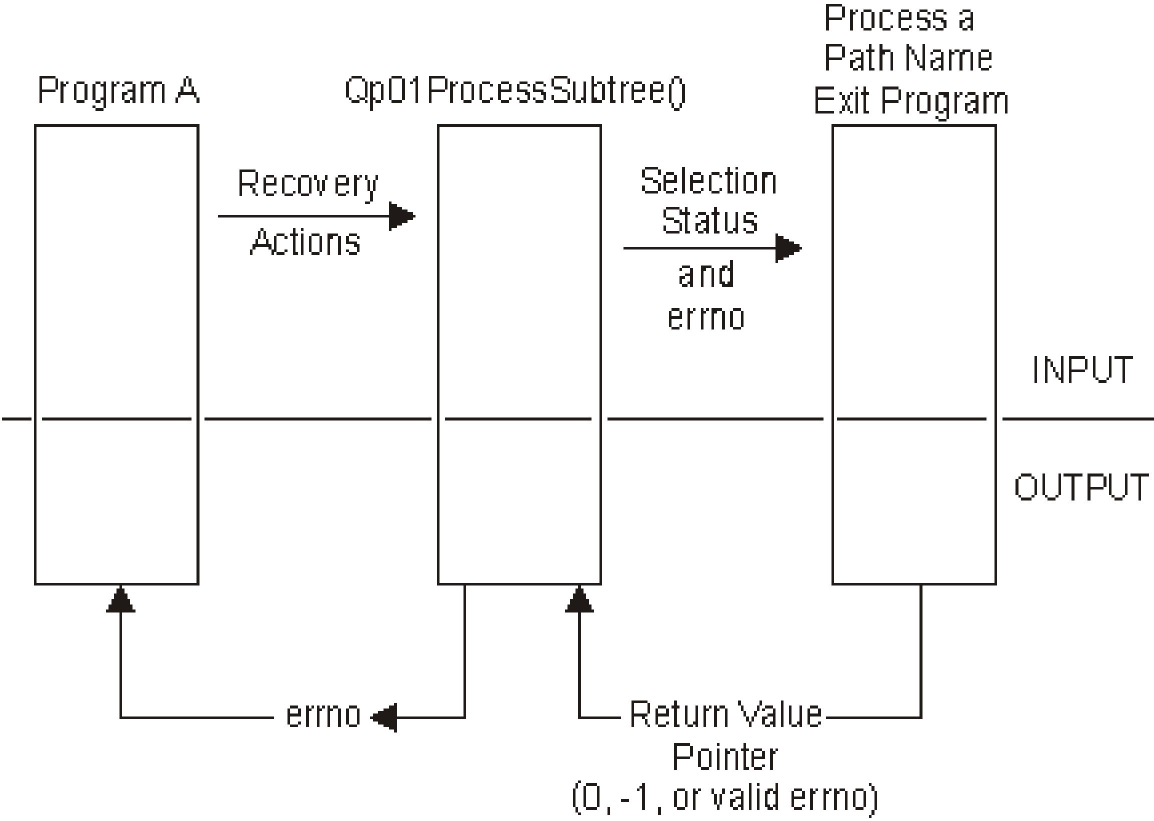
  There are three separate parameters that control error recovery during a search. The caller of the API determines how an error should be reported to the exit program by setting the *Err_recovery_actions* parameter. The API sets the *Selection status pointer* and sends it to the exit program to indicate one of four conditions: the API search status is OK, the last object has been processed, the API has encountered recoverable errors, or the search cannot continue. For error conditions it also sends a valid *errno*. The exit program returns an indicator back to the API either to continue or to end the search by setting the *Return value pointer*. For error conditions, it also returns a valid *errno*, pointed to by the *Return value pointer*. Each time **Qp0lProcessSubtree()** regains control from the exit program, it determines whether the search should continue or end by evaluating the *Err_recovery_actions* parameter, its *Selection status pointer*, and the *Return value pointer*. Upon ending, **Qp0lProcessSubtree()** returns 0 to indicate a successful search, or a -1 and an *errno* to indicate the error condition. This *errno* may have been set by the exit program (*Return value pointer*).

  This error recovery design allows for flexibility in handling errors between the caller, the API, and the exit program. Whenever an unrecoverable error occurs, if possible, the exit program is given a final call; this call allows the exit program to do such tasks as cleanup or to put information in the function control block, or to record information about the error. However, the exit program cannot decide that the search should continue. The API will return to its caller when it regains control. There are only two specific instances in which the API determines that the exit program is not called:

  – When the API cannot resolve the exit program name or its authorization.

  – When input parameters are missing or specified incorrectly. (The API returns EINVAL to the caller before any other processing.)

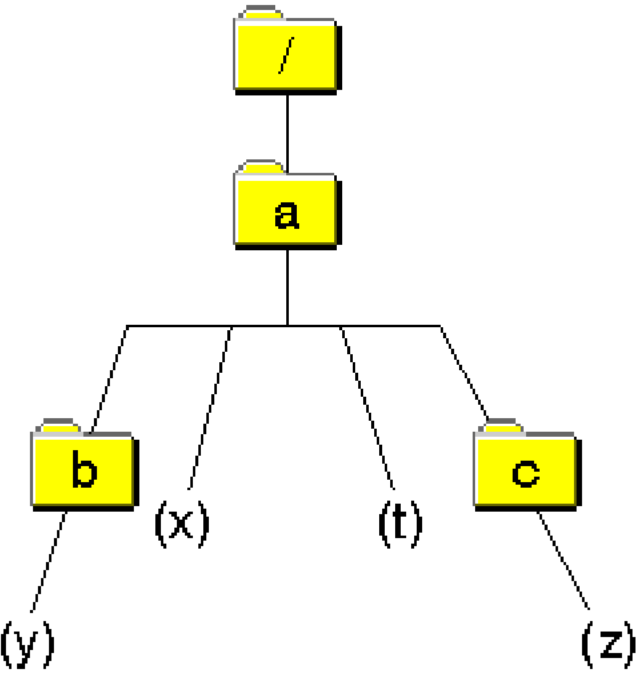
  Following is a diagram showing the flow and relationship of these parameters.



Scenarios

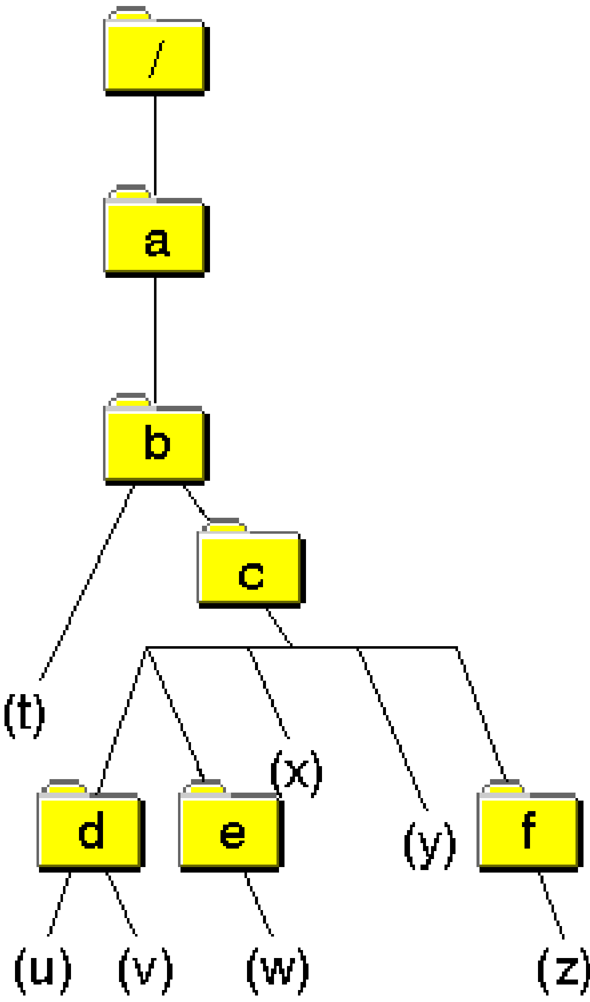
Following are scenarios showing calls and the results of calls to **Qp0lProcessSubtree()**. "Figure: Directory Structure A" on page 365 and "Figure: Directory Structure B" on page 366 define the input directory structure for these scenarios.

Figure: Directory Structure A



This directory structure represents three subdirectories (a, b, c), three objects (x, y, z), and a symbolic link (t).

Figure: Directory Structure B



This directory structure represents six subdirectories (a, b, c, d, e, f) and seven objects (t, u, v, w, x, y, z).

Scenario 1

This scenario assumes processing a directory as shown by *Directory Structure A* in "Figure: Directory Structure A" on page 365.

This scenario shows a call to the API without any criteria to filter the selection of objects in the path being searched. If the API call were coded with the parameter values as shown by *Input value* in "Figure: Scenario 1 API Input," the exit program would be called nine times and would pass the object names as shown by the *Object Name Pointer* in "Figure: Results of a call" on page 367. Because QP0L_SUBTREE_YES is specified, all of the directories in the path will be opened and the name of all the objects that they contain will be passed to the exit program. Note that the only guaranteed order is that parent directories are passed to the exit program after all of their children.

Figure: Scenario 1 API Input

Input Parameter	Input value
*Path_Name	'/' ('/' processes every directory on the system and is not recommended if performance is a consideration)
Subtree_level	QP0L_SUBTREE_YES

Input Parameter	Input value
*Objtypes_array_ptr	NULL
Local_remote_obj	QP0L_LOCAL_REMOTE_OBJ
*IN_EXclusion_ptr	NULL
Err_recovery_action	QP0L_PASS_WITH_ERRORID
*UserFunction_ptr	QP0L_USER_FUNCTION_PTR
*Function_CtlBlk_ptr	NULL

Figure: Results of a call

Exit Program Call Count	Object Name Pointer
1	/a/b/y
2	/a/b
3	/a/x
4	/a/t
5	/a/c/z
6	/a/c
7	/a
8	/
9	NULL path name (indicates the API completed)

Scenario 2

This scenario assumes processing a directory as shown by *Directory Structure A* in the “Figure: Directory Structure A” on page 365.

This shows a call to the API with the *Subtree level* parameter set to retrieve only one level, without any object filtering. Since QP0L_SUBTREE_NO is specified, the names of all objects in the path will be passed to the exit program, however, none of the directories will be opened. This allows a caller to perform tasks such as identifying all of the root objects for a file system. For example, this would identify all of the first level folders, when processing against the QDLS file system. Then the API can be called recursively from within the exit program, with each of these folders specified as the path to be searched.

If the API call were coded with the parameter values as shown by *Input value* in “Figure: Scenario 2 API Input,” the exit program would be called six times and would pass the object names as shown by the *Object Name Pointer* in “Figure: Results of a call” on page 368.

Figure: Scenario 2 API Input

Input Parameter	Input value
*Path_Name	’/a’
Subtree_level	QP0L_SUBTREE_NO
*Objtypes_array_ptr	NULL
Local_remote_obj	QP0L_LOCAL_REMOTE_OBJ
*IN_EXclusion_ptr	NULL

Input Parameter	Input value
Err_recovery_action	QP0L_PASS_WITH_ERRORID
*UserFunction_ptr	QP0L_USER_FUNCTION_PTR
*Function_CtlBlk_ptr	NULL

Figure: Results of a call

Exit Program Call Count	Object Name Pointer
1	/a/b
2	/a/x
3	/a/t
4	/a/c
5	/a
6	NULL path name (indicates the API completed)

Scenario 3

This scenario assumes processing a directory as shown by *Directory Structure B* in the “Figure: Directory Structure B” on page 366.

This scenario represents a call to the API with an inclusion list. Note that the *Path Name* parameter is not used as the starting directory since each entry in an inclusion list is treated as a starting directory.

If the API call were coded with the parameter values as shown by *Input value* in “Figure: Scenario 3 API Input,” the exit program would be called six times and would pass the object names as shown by the *Object Name Pointer* in “Figure: Results of a call” on page 369.

Note that /a/b/c/d/v could be returned before /a/b/c/d/u, as shown in this scenario, since children in a directory can be returned in any order. The only guaranteed order is that the exit program is called with all children objects before being called with the parent to allow the exit program to delete directories if desired.

Figure: Scenario 3 API Input

Input Parameter	Input value
*Path_Name	NULL (not used with an inclusion list)
Subtree_level	QP0L_SUBTREE_YES
*Objtypes_array_ptr	'*DIR ' '*STMF '
Local_remote_obj	QP0L_LOCAL_OBJ
*IN_EXclusion_ptr	QP0L_INCLUSION_TYPE, '/a/b/c/d/' '/a/b/c/e/'
Err_recovery_action	QP0L_PASS_WITH_ERRORID
*UserFunction_ptr	QP0L_USER_FUNCTION_PTR
*Function_CtlBlk_ptr	NULL

Figure: Results of a call

Exit Program Call Count	Object Name Pointer
1	/a/b/c/d/v
2	/a/b/c/d/u
3	/a/b/c/d
4	/a/b/c/e/w
5	/a/b/c/e/
6	NULL path name (indicates the API completed)

Scenario 4

This scenario assumes processing a directory as shown by *Directory Structure B* in the "Figure: Directory Structure B" on page 366.

This scenario represents a call to the API with an exclusion list. Note that each relative entry in the exclusion list is resolved relative to the current working directory at the time the API is called. This scenario assumes that the current working directory is /a/b/.

If the API call were coded with the parameter values as shown by *Input value* in "Figure: Scenario 4 API Input," the exit program would be called eight times and would pass the object names as shown by the *Object Name Pointer* in "Figure: Results of a call."

This scenario also shows that children in a directory can be returned in any order. The only guaranteed order is that the exit program is called with all children objects before being called with the parent to allow the exit program to delete directories if desired.

Figure: Scenario 4 API Input

Input Parameter	Input value
*Path_Name	'/a/b/'
Subtree_level	QP0L_SUBTREE_YES
*Objtypes_array_ptr	'*DIR ' '*STMF '
Local_remote_obj	QP0L_LOCAL_OBJ
*IN_EXclusion_ptr	QP0L_EXCLUSION_TYPE, 'c/d/' 'c/e/'
Err_recovery_action	QP0L_PASS_WITH_ERRORID
*UserFunction_ptr	QP0L_USER_FUNCTION_PTR
*Function_CtlBlk_ptr	NULL

Figure: Results of a call

Exit Program Call Count	Object Name Pointer
1	/a/b/t
2	/a/b/c/y
3	/a/b/c/f/z

Exit Program Call Count	Object Name Pointer
4	/a/b/c/f
5	/a/b/c/x
6	/a/b/c
7	/a/b
8	NULL path name (indicates the API completed)

Related Information

- The <**Qp0lstdi.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**qlg.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "QlgProcessSubtree()—Process a Path Name (using NLS-enabled path name)" on page 279—Process a Path Name (using NLS-enabled path name)
- "Process a Path Name Exit Program" on page 533

Example

See Code disclaimer information for information pertaining to code examples.

Following is a code example showing a call to the Qp0lProcessSubtree() API with a procedure as the exit program:

```
/*****************************************************************/
/*****************************************************************/

#include <Qp0lstdi.h>
#include <stdio.h>
#include <errno.h>
#include <qtqiconv.h>

void Obj_Print_Function
                    (uint            *Selection_status_pointer,
                     uint            *Error_value_pointer,
                     uint            *Return_value_pointer,
                     Qlg_Path_Name_T *Object_name_pointer,
                     void            *Function_control_block_pointer)
{
 /*****************************************************************/
 /* This exit program example prints the names, one at a time,   */
 /* of each entry in a directory structure that it receives on   */
 /* each call from Qp0lProcessSubtree().                         */
 /*****************************************************************/

 #define PATH_TYPE_POINTER    0x00000001  /* If this flag is on, */
                             /* the qlg structure contains a    */
                             /* pointer to the path name.       */
                             /* Otherwise, the path name is in  */
                             /* contiguous storage within the   */
                             /* qlg structure.                  */

 typedef union pn_input_type
 {
    char pn_char_type[256];     /* path name is in    */
                                          /* contiguous storage */
    char *pn_ptr_type;       /* path name is a pointer          */
 };
typedef struct pnstruct
    {
```

```
    Qlg_Path_Name_T qlg_struct;
    union pn_input_type   pn;
   };
struct pnstruct *pns;
char *path_ptr;

size_t insz;
size_t outsz = 1000;
char outbuf[1000];
char *outbuf_ptr;
iconv_t cd;
size_t ret_iconv;

QtqCode_T toCode   =    {37,0,0,0,0,0};
QtqCode_T fromCode = {61952,0,0,1,0,0};

if (*Selection_status_pointer == QP0L_SELECT_OK)
 {
  if (Object_name_pointer != NULL)
   {
    /*********************************************************/
    /* Point to the pathname and get the size of the pathname   */
    /* that was sent from the Qp0lProcessSubtree() API.  The    */
    /* format of the pathname must be determined by evaluating  */
    /* Path_Type in the qlg structure.                          */
    /*********************************************************/
    pns = (struct pnstruct *)Object_name_pointer;
    if (Object_name_pointer->Path_Type & PATH_TYPE_POINTER)
    {
      path_ptr = pns->pn.pn_ptr_type;
    }
    else
    {
      path_ptr = (char *)(pns->pn.pn_char_type);
    }
    insz = pns->qlg_struct.Path_Length;


    /*********************************************************/
    /* Initialize the print buffer.                             */
    /*********************************************************/
    outbuf_ptr = (char *)outbuf;
    memset(outbuf_ptr, 0x00, insz);


    /*********************************************************/
    /* Use iconv to convert from 61952 to the job CCSID.        */
    /* REMEMBER iconv will change the data that it receives.    */
    /*********************************************************/
    cd = /* Open the conversion descriptor.*/
         QtqIconvOpen(&toCode,
                      &fromCode);
    if (cd.return_value == -1)
      {
       /******************************************************/
       /* If conversion descriptor was not opened successfully, */
       /* return an error and errno (ECONVERT) to the API.      */
       /******************************************************/
       *Return_value_pointer = errno;
       return;
       }

    ret_iconv = /* Perform the conversion.*/
                (iconv(cd,
                       (char **)&(path_ptr),
                       &insz,
                       (char **)&(outbuf_ptr),
```

```
                      &outsz));
     if (ret_iconv != 0)
       {
        /*******************************************************/
        /* If the conversion failed, close the conversion       */
        /* descriptor and return an error and errno (ECONVERT)   */
        /* to the API.                                          */
        /*******************************************************/
        ret_iconv= iconv_close(cd);
        *Return_value_pointer = errno;
        return;
        }

     /************************************************************/
     /* Print the name of the object being processed and close   */
     /* the conversion descriptor.                                */
     /************************************************************/
     printf("In User Exit Program. Path is %s.\n", outbuf);
     ret_iconv = iconv_close(cd);

   }                         /* end Object_name_pointer != NULL */
   else
      {
       printf("In User Exit Program with a null Pathname \n");
      }
  }               /* end *Selection_status_pointer == QP0L_SELECT_OK */

     *Return_value_pointer = 0;

} /* end Exit program */



int main (int argc, char *argv[])
  {
   #define MYPN "/TestDir"
   const int  zero = 0;
   const char US_const[3]= "US";
   const char Language_const[4]="ENU";
   const char Path_Name_Del_const[2]= "/";
   const char LibObj_const[12]= "*LIB       ";
   typedef struct pnstruct
    {
     Qlg_Path_Name_T  qlg_struct;
     char            pn[50];  /* Must be greater than  */
                                        /* or equal the length   */
                                        /* of the path name.     */
     };
   struct pnstruct pns;
   Qp0l_Objtypes_List_t   MyObj_types;
   Qp0l_User_Function_t   User_function;
   struct
     {
      uint    AnyData_to_the_exitprogram;
      uint    AnyData_not_processed_by_the_API;
     } CtlBlkAreaName;

   int rc;
  /****************************************************************/
  /* In this example, the pathname is defined by MYPN as TestDir */
  /* and it is assumed that the TestDir directory exists on the  */
  /* system.  Various other functions or other routines could be */
  /* included here to (for example):                             */
  /*    1) determine the beginning search directory.             */
  /*    2) construct the path name in the correct format.        */
  /*    3) others...                                             */
  /****************************************************************/
```

```
 /*************************************************************/
 /*************************************************************/
 /*   Initialize Qp0lProcessSubtree() API Parameters          */
 /*************************************************************/
 memset((void*)&pns, 0x00, sizeof(struct pnstruct));
 pns.qlg_struct.CCSID = 37;
 memcpy(pns.qlg_struct.Country_ID,US_const,2);
 memcpy(pns.qlg_struct.Language_ID,Language_const,3);
 pns.qlg_struct.Path_Type = zero;
 pns.qlg_struct.Path_Length = sizeof(MYPN)-1;
 memcpy(pns.qlg_struct.Path_Name_Delimiter,Path_Name_Del_const,1);
 memcpy(pns.pn,MYPN,sizeof(MYPN));
 MyObj_types.Number_Of_Objtypes = zero;
 memset((void *)&User_function, 0x00, sizeof(Qp0l_User_Function_t));
 User_function.Function_Type = QP0L_USER_FUNCTION_PTR;
 User_function.Mltthdacn[0] = QP0L_MLTTHDACN_NOMSG;
 User_function.Procedure = &Obj_Print_Function;

if (rc = Qp0lProcessSubtree((Qlg_Path_Name_T *)&pns,
                            QP0L_SUBTREE_YES,
                            (Qp0l_Objtypes_List_t *)NULL,
                            QP0L_LOCAL_REMOTE_OBJ,
                            (Qp0l_IN_EXclusion_List_t *)NULL,
                            QP0L_PASS_WITH_ERRORID,
                            &User_function,
                            &CtlBlkAreaName) == 0)
  {
    printf("Qp0lProcessSubtree() Successful : error = %d\n", errno);
  }
else
    {/*unsuccessful return from Qp0lProcessSubtree() API  */
     printf("ERROR on Qp0lProcessSubtree(): error = %d\n", errno);
     perror("Error message");
    }
 }       /* end main */
```

API introduced: V4R3

Top | UNIX-Type APIs | APIs by category

Qp0lRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists

Syntax

```
#include <Qp0lstdi.h>

int Qp0lRenameKeep(const char *old, const char *new);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 377.

The **Qp0lRenameKeep()** function renames a file or a directory specified by *old* to the name given by *new*. The *old* pointer must specify the name of an existing file or directory. Both *old* and *new* must be of the same type; that is, both directories or both files. The last element of *old* and *new* must not be ″dot″ (.) or ″dot-dot″ (..).

If *new* already exists, **Qp0lRenameKeep()** fails with the [EEXIST] error.

If the *old* argument points to a symbolic link, the symbolic link is renamed. **Qp0lRenameKeep()** does not affect any file or directory named by the contents of the symbolic link. See “Usage Notes” on page 377 for more information.

When **Qp0lRenameKeep()** is successful, it updates the change and modification times for the parent directories of *old* and *new*.

If the *old* object is checked out, **Qp0lRenameKeep()** fails with the [EBUSY] error.

Parameters

old (Input) A pointer to the null-terminated path name of the file to be renamed.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See “QlgRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists (using NLS-enabled path name)” on page 286 for a description and an example of supplying the *old* in any CCSID.

new (Input) A pointer to the null-terminated path name of the new name of the file.

This parameter is assumed to be represented in the CCSID currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

The new file name is assumed to be represented in the language and country or region currently in effect for the job.

See “QlgRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists (using NLS-enabled path name)” on page 286 for a description and an example of supplying the *new* in any CCSID.

Authorities

Note: Adopted authority is not used.

Authorization Required for Qp0lRenameKeep() (excluding QSYS.LIB, independent ASP QSYS.LIB, QDLS, and QOPT)

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object	*WX	EACCES
old object if it is a directory	*OBJMGT + *W	EACCES
old object if it is not a directory	*OBJMGT	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object	*WX	EACCES
Parent directory of the *old* object has the S_ISVTX mode bit set to binary one (see **Note**).	*ALLOBJ, or owner of the *old* object, or owner of the parent directory of the *old* object	EPERM

Note: The S_ISVTX mode bit (which is equivalent to the ’Restricted rename and unlink’ object attribute) restriction only applies to objects in the ″root″ (/), QOpenSys, and user-defined file systems.

Authorization Required for Qp0lRenameKeep() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object if the object is a database file member	*OBJMGT	EACCES
Parent directory of the parent directory of *old* object if the object is a database file member	*UPD	EACCES
Parent directory of *old* object if the object is not a database file member	*RWX	EACCES
old object if it is a database file member	None	None
old object if it is not a database file member	*OBJMGT	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object if object is not a database file member	*RWX	EACCES

Authorization Required for Qp0lRenameKeep() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object	*CHANGE	EACCES
old object	*ALL	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object	*CHANGE	EACCES

Authorization Required for Qp0lRenameKeep() in the QOPT File System

Object Referred to	Authority Required	errno
Volume authorization list for volume to be renamed in a media library device	*ALL	EACCES
Volume authorization list for volume to be renamed in a stand alone device	*CHANGE	EACCES
Volume authorization list for volume containing object to be renamed	*CHANGE	EACCES
Root directory (/) of volume to be renamed if volume media format is Universal Disk Format (UDF)	*RWX	EACCES
Each directory in old path name preceding the object to be renamed if volume media format is Universal Disk Format (UDF)	*X	EACCES
Parent directory of old object if volume media format is Universal Disk Format (UDF)	*WX	EACCES
Old object if volume media format is Universal Disk Format (UDF)	*W	EACCES
Each directory in new path name preceding the object if volume media format is Universal Disk Format (UDF)	*X	EACCES
Parent directory of new object if volume media format is Universal Disk format (UDF)	*WX	EACCES
Object and parent directories if volume media format is not Universal Disk format (UDF)	None	None

Return Value

0 **Qp0lRenameKeep()** was successful.

-1 **Qp0lRenameKeep()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **Qp0lRenameKeep()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EDATALINK (page 547)]	
[EEXIST (page 543)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	For example, may be returned if the directories preceding the object to be renamed in the *old* path name are part of *new*, or if either name refers to dot or dot-dot.
[EIO (page 540)]	
[EISDIR (page 544)]	*new* is a directory, but *old* is not a directory.
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[EMLINK (page 544)]	*old* is a directory and the link count of the parent directory of *new* would exceed LINK_MAX.
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	
[EXDEV (page 541)]	*old* and *new* identify files or directories in different file systems. Links between different file systems are not allowed.

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. About the Rename Functions

   The integrated file system provides two functions that rename a file or directory. Both rename the *old* path name to a *new* path name. The difference is determined by what happens when *new* already exists:
   - If *new* already exists when using **Qp0lRenameKeep()**, the rename fails with the [EEXIST] error.

- If *new* already exists when using **Qp0lRenameUnlink()**, the existing path name is unlinked (removed) before *old* is renamed to *new*.

These functions are defined in the **<Qp0lstdi.h>** header file. When **<Qp0lstdi.h>** is included, the **rename()** function is defined to be either **Qp0lRenameUnlink()** or **Qp0lRenameKeep()**, depending on the definitions of the _POSIX_SOURCE and _POSIX1_SOURCE macros:

- When _POSIX_SOURCE and _POSIX1_SOURCE are **not** defined, **rename()** is defined to be **Qp0lRenameKeep()**. Either **rename()** or **Qp0lRenameKeep()** can be used to rename a file or directory with the semantics of **Qp0lRenameKeep()**.
- When _POSIX_SOURCE or _POSIX1_SOURCE is defined, **rename()** is defined to be **Qp0lRenameUnlink()**. Either **rename()** or **Qp0lRenameUnlink()** can be used to rename a file or directory with the semantics of **Qp0lRenameUnlink()**.

When the **<Qp0lstdi.h>** header file is **not** included, **rename()** operates only on database files in the QSYS.LIB and independent ASP QSYS.LIB file systems, as it did before the introduction of the integrated file system.

3. QSYS.LIB and Independent ASP QSYS.LIB File System Differences
   - When a database member is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, the sequence of ″directories″ (library and file) preceding the object in the *new* path name must be the same as the sequence of directories preceding the object in the *old* path name.
   - The following object types cannot be renamed when there are secondary threads active in the job: *CFGL, *CNNL, *CTLD, *DEVD, *LIND, *NWID. The operation will fail with error code [ENOTSAFE].
   - When a library is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, the sequence of ″directories″ (/QSYS.LIB or /asp_name/QSYS.LIB, where asp_name is the independent Auxiliary Storage Pool name) preceding the object in the *new* path name must be the same as the sequence of directories preceding the object in the *old* path name [EINVAL].
4. QDLS File System Differences

   When a folder is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, a folder must remain in the same parent folder.
5. QOPT File System Differences

   You can rename only a volume or a file, not a directory.
6. QFileSvr.400 File System Differences

   You cannot rename the first-level directory. For example, you cannot rename `Dir1` in the path name `/QFileSvr.400/Dir1/Dir2/Object`. The first-level directory identifies the target system in a communications connection.
7. QNetWare File System Differences

   The new and old files or directories must exist on the same NetWare server. This function cannot be used to move data from one server to another.
8. QNTC File System Differences

   The *new* and the *old* files or directories must exist on the same Windows NT server. This function cannot be used to move data from one server to another.
9. ″Root″ (/) and User-defined File System Differences

   If the file being renamed is in the ″root″ (/) file system or in a monocase user-defined file system, and the file system has the *TYPE2 directory format, and both *old* and *new* refer to the same link, but their case is different (eg. /ABC and /Abc), Qp0lRenameUnlink() changes the link name to the *new* name.
10. The file cannot be renamed if the file is a DataLink column in an SQL table and a row in that SQL table references this file.

Related Information

- The <**stdio.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**Qp0lstdi.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "pathconf()—Get Configurable Path Name Variables" on page 216—Get Configurable Path Name Variables
- "rename()—Rename File or Directory" on page 460—Rename File or Directory
- "QlgRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists (using NLS-enabled path name)" on page 286—Rename File or Directory, Keep ″new″ If It Exists (using NLS-enabled path name)
- "Qp0lRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists"—Rename File or Directory, Unlink ″new″ If It Exists

Example

See Code disclaimer information for information pertaining to code examples.

When you pass two file names to this example, it will try to change the file name from the first name to the second using **Qp0lRenameKeep()**.

```
#include <Qp0lstdi.h>

int main(int argc, char ** argv ) {

  if ( argc != 3 )
    printf( "Usage: %s old_fn new_fn\n", argv[0] );
  else if ( Qp0lRenameKeep( argv[1], argv[2] ) != 0
    perror ( "Could not rename file" );
}
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

Qp0lRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists

Syntax

```
#include <Qp0lstdi.h>

int Qp0lRenameUnlink(const char *old, const char *new);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 383.

The **Qp0lRenameUnlink()** function renames a file or a directory specified by *old* to the name given by *new*. The *old* pointer must specify the name of an existing file or directory. Both *old* and *new* must be of the same type; that is, both directories or both files. The last element of *old* and *new* must not be ″dot″ (.) or ″dot-dot″ (..).

If *new* already exists, it is removed before *old* is renamed to *new*. Therefore, if *new* specifies the name of an existing directory, the directory must be empty.

If the *old* argument points to a symbolic link, the symbolic link is renamed. If the *new* argument points to a symbolic link, the link is removed and *old* is renamed to *new*. **Qp0lRenameUnlink()** does not affect any file or directory named by the contents of the symbolic link.

If *old* and *new* both refer to the same file, **Qp0lRenameUnlink()** returns successfully and performs no other action. See “Usage Notes” on page 383 for more information.

When **Qp0lRenameUnlink()** is successful, it updates the change and modification times for the parent directories of *old* and *new*.

If the *old* object is checked out, **Qp0lRenameUnlink()** fails with the [EBUSY] error.

Parameters

old (Input) A pointer to the null-terminated path name of the file to be renamed.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See “QlgRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists (using NLS-enabled path name)” on page 288 for a description and an example of supplying the *old* in any CCSID.

new (Input) A pointer to the null-terminated path name of the new name of the file.

This parameter is assumed to be represented in the CCSID currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

The new file name is assumed to be represented in the language and country or region currently in effect for the process.

See “QlgRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists (using NLS-enabled path name)” on page 288 for a description and an example of supplying the *new* in any CCSID.

Authorities

Note: Adopted authority is not used.

Authorization Required for Qp0lRenameUnlink() (excluding QSYS.LIB, independent ASP QSYS.LIB, QDLS, and QOPT)

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object	*WX	EACCES
old object if it is a directory	*OBJMGT + *W	EACCES
old object if it is not a directory	*OBJMGT	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object	*WX	EACCES
New object, if it exists	*OBJEXIST	EACCES
Parent directory of the *old* object has the S_ISVTX mode bit set to binary one (see **Note**).	*ALLOBJ, or owner of the *old* object, or owner of the parent directory of the *old* object	EPERM
Parent directory of the *new* object, if it exists, has the S_ISVTX mode bit set to binary one (see **Note**).	*ALLOBJ, or owner of the *new* object, or owner of the parent directory of the *new* object	EPERM

Object Referred to	Authority Required	errno
Note: The S_ISVTX mode bit (which is equivalent to the 'Restricted rename and unlink' object attribute) restriction only applies to objects in the "root" (/), QOpenSys, and user-defined file systems.		

Authorization Required for Qp0lRenameUnlink() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object if the object is a database file member	*OBJMGT	EACCES
Parent directory of the parent directory of *old* object if the object is a database file member	*UPD	EACCES
Parent directory of *old* object if the object is not a database file member	*RWX	EACCES
old object if it is a database file member	None	None
old object if it is not a database file member	*OBJMGT, Ownership required if *new* object already exists	EACCES
old object if it is a *FILE object type	*OBJMGT, *OBJOPR, Ownership required if *new* object already exists	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object if object is not a database file member	*RWX	EACCES
new object if object already exists and is not a database file member, *PGM, *MENU, *FILE, *LIB, or *SBSD object type	*OBJEXIST, *OBJMGT	EACCES
new object if object already exists and is a *PGM object type	*OBJEXIST, *OBJMGT, *READ	EACCES
new object if object already exists and is a *MENU or *FILE object type	*OBJEXIST, *OBJOPR	EACCES
new object if object already exists and is a *LIB or *SBSD object type	*OBJEXIST, *OBJMGT, *RX	EACCES

Authorization Required for Qp0lRenameUnlink() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in *old* path name preceding the object to be renamed	*X	EACCES
Parent directory of *old* object	*CHANGE	EACCES
old object	*ALL	EACCES
Each directory in *new* path name preceding the object	*X	EACCES
Parent directory of *new* object	*CHANGE	EACCES

Authorization Required for Qp0lRenameUnlink() in the QOPT File System

Object Referred to	Authority Required	errno
Volume to be renamed	*ALL	EACCES
Volume containing object to be renamed	*CHANGE	EACCES

Object Referred to	Authority Required	errno
Object within volume	None	None

Return Value

0 **Qp0lRenameUnlink()** was successful.

-1 **Qp0lRenameUnlink()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **Qp0lRenameUnlink()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EDATALINK (page 547)]	
[EEXIST (page 543)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	For example, may be returned if the directories preceding the object to be renamed in the *old* path name are part of *new*, or if either name refers to dot or dot-dot.
[EIO (page 540)]	
[EISDIR (page 544)]	*new* is a directory, but *old* is not a directory.
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOTAVAIL (page 547)]	
[ENOTEMPTY (page 545)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EMLINK (page 544)]	*old* is a directory and the link count of the parent directory of *new* would exceed LINK_MAX.

Error condition	Additional information
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	
[EXDEV (page 541)]	*old* and *new* identify files or directories in different file systems. Links between different file systems are not allowed.

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB

- QOPT
- Network File System
- QFileSvr.400

2. About the Rename Functions

   The integrated file system provides two functions that rename a file or directory. Both rename the *old* path name to a *new* path name. The difference is determined by what happens when *new* already exists:

   - If *new* already exists when using **Qp0lRenameUnlink()**, the existing path name is unlinked (removed) before *old* is renamed to *new*.
   - If *new* already exists when using **Qp0lRenameKeep()**, the rename fails with the [EEXIST] error.

   These functions are defined in the **<Qp0lstdi.h>** header file. When **<Qp0lstdi.h>** is included, the **rename()** function is defined to be either **Qp0lRenameUnlink()** or **Qp0lRenameKeep()**, depending on the definitions of the _POSIX_SOURCE and _POSIX1_SOURCE macros:

   - When _POSIX_SOURCE or _POSIX1_SOURCE is defined, **rename()** is defined to be **Qp0lRenameUnlink()**. Either **rename()** or **Qp0lRenameUnlink()** can be used to rename a file or directory with the semantics of **Qp0lRenameUnlink()**.
   - When _POSIX_SOURCE and _POSIX1_SOURCE are **not** defined, **rename()** is defined to be **Qp0lRenameKeep()**. Either **rename()** or **Qp0lRenameKeep()** can be used to rename a file or directory with the semantics of **Qp0lRenameKeep()**.

   When the **<Qp0lstdi.h>** header file is **not** included, **rename()** operates only on database files in the QSYS.LIB and independent ASP QSYS.LIB file systems, as it did before the introduction of the integrated file system.

3. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   - When a database member is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, the sequence of "directories" (library and file) preceding the object in the *new* path name must be the same as the sequence of directories preceding the object in the *old* path name. If *new* already exists, [ENOTSUP] is returned.
   - The following object types cannot be renamed when there are secondary threads active in the job: *CFGL, *CNNL, *CTLD, *DEVD, *LIND, *NWID. The operation will fail with error code [ENOTSAFE].
   - When a library is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, the sequence of "directories" (/QSYS.LIB or /asp_name/QSYS.LIB, where asp_name is the independent Auxiliary Storage Pool name) preceding the object in the *new* path name must be the same as the sequence of directories preceding the object in the *old* path name.

4. QDLS File System Differences

   When a folder is being renamed, the part of the *new* path name preceding the object must be the same as that of the *old* path name. That is, a folder must remain in the same parent folder.

   If the object identified by the *new* path name exists, QDLS returns the [EEXIST] error.

5. QOPT File System Differences

   You can rename only a volume or a file, not a directory.

   If the object identified by the new path name exists, QOPT returns the [EEXIST] error.

6. QFileSvr.400 File System Differences

   You cannot rename the first-level directory. For example, you cannot rename `Dir1` in the path name `/QFileSvr.400/Dir1/Dir2/Object`. The first-level directory identifies the target system in a communications connection.

7. QNetWare File System Differences

   The new and old files or directories must exist on the same NetWare server. This function cannot be used to move data from one server to another.

8. QNTC File System Differences

The *new* and the *old* files or directories must exist on the same Windows NT server. This function cannot be used to move data from one server to another.

9. ″Root″ (/) and User-defined File System Differences

   If the file being renamed is in the ″root″ (/) file system or in a monocase user-defined file system, and the file system has the *TYPE2 directory format, and both *old* and *new* refer to the same link, but their case is different (eg. /ABC and /Abc), Qp0lRenameUnlink() changes the link name to the *new* name.

10. The file cannot be renamed if the file is a DataLink column in an SQL table and a row in that SQL table references this file.

Related Information

- The <**stdio.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**Qp0lstdi.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "pathconf()—Get Configurable Path Name Variables" on page 216—Get Configurable Path Name Variables
- "rename()—Rename File or Directory" on page 460—Rename File or Directory
- "Qp0lRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists" on page 373—Rename File or Directory, Keep ″new″ If It Exists
- "QlgRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists (using NLS-enabled path name)" on page 288—Rename File or Directory, Unlink ″new″ If It Exists (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

When you pass two file names to this example, it will try to change the file name from the first name to the second using **Qp0lRenameUnlink()**.

```
#include <Qp0lstdi.h>

int main(int argc, char ** argv ) {

  if ( argc != 3 )
    printf( "Usage: %s old_fn new_fn\n", argv[0] );
  else if ( Qp0lRenameUnlink( argv[1], argv[2] ) != 0
    perror ( "Could not rename file" );
}
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

Retrieve Object References (QP0LROR)

Syntax

```
#include <qp0lror.h>
void QP0LROR(
              void *           Receiver_Ptr,
              unsigned int     Receiver_Length,
              char *           Format_Ptr,
              Qlg_Path_Name_T * Path_Ptr,
              void *           Error_Code_Ptr

             );
```

Default Public Authority: *USE

Threadsafe: No

The **QP0LROR()** API is used to retrieve information about integrated file system references on an object.

A reference is an individual type of access or lock obtained on the object when using integrated file system interfaces. An object may have multiple references concurrently held, provided that the reference types do not conflict with one another.

This API will not return information about byte range locks that may currently be held on an object.

Parameters

Receiver_Ptr
(Output)

The variable that is to receive the information requested. You can specify the size of this area to be smaller than the format requested as long as you specify the length parameter correctly. As a result, the API returns only the data that the area can hold.

The format of the output is described by either the RORO0100 output format or the RORO0200 output format. See "RORO0100 Output Format Description (*Qp0l_RORO0100_Output*)" on page 387 or the "RORO0200 Output Format Description (*Qp0l_RORO0200_Output*)" on page 387 for a detailed description of these output formats.

Receiver_Length
(Input)

The length of the receiver variable. If the length is larger than the size of the receiver variable, the results may not be predictable. The minimum length is 8 bytes.

Format_Ptr
(Input)

Pointer to a 8 byte character string that identifies the desired output format. It must be one of the following values:

RORO0100
The reference type output will be formatted in a RORO0100 format. See "RORO0100 Output Format Description (*Qp0l_RORO0100_Output*)" on page 387. This format gives the caller a quick view of the object's references.

RORO0200
The reference type output will be formatted in a RORO0200 format. See "RORO0200 Output Format Description (*Qp0l_RORO0200_Output*)" on page 387. Specifying this format may cause QP0LROR to be a long running operation. The length of time it will take to complete depends on the number of jobs active on the system, and the number of jobs currently using objects through integrated file system interfaces.

Path_Ptr
(Input)

Pointer to the path name to the object whose reference information is to be obtained. The path name must be specified in an NLS-enabled format specified by the Qlg_Path_Name structure. For more information on the Qlg_Path_Name_T structure, see Path name format.

If the last element of the *path* is a symbolic link, the **Qp0lROR()** function does not resolve the contents of the symbolic link. The reference information will be obtained for the symbolic link itself.

Error_Code_Ptr
(Input/Output)

Pointer to an error code structure to receive error information. See Error Code Parameter for more information.

Authorities and Locks

Directory Authority
The user must have execute (*X) data authority to each directory preceding the object whose references are to be obtained.

Object Authority
The user must have read (*R) data authority to the object whose references are to be obtained.

Output Structure Formats

RORO0100 Output Format Description (*Qp0l_RORO0100_Output*)

This structure is used to return object reference information.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4), UNSIGNED	Bytes Returned
4	4	BINARY(4), UNSIGNED	Bytes Available
8	8	BINARY(4), UNSIGNED	Offset to Simple Reference Types
12	0C	BINARY(4), UNSIGNED	Length of Simple Reference Types
16	10	BINARY(4), UNSIGNED	Reference Count
20	14	BINARY(4), UNSIGNED	In-Use Indicator
Offset determined from *Offset to Simple Reference Types* field		Qp0l_Sim_Ref_Types_Output Structure	Simple Reference Types Structure. See "Simple Object Reference Types Structure Description (*Qp0l_Sim_Ref_Types_Output*)" on page 389 for a description of this structure.

RORO0200 Output Format Description (*Qp0l_RORO0200_Output*)

This output format is used to return object reference information, including a list of jobs known to be referencing the object. This includes everything from the RORO0100 structure plus additional information.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4), UNSIGNED	Bytes Returned
4	4	BINARY(4), UNSIGNED	Bytes Available
8	8	BINARY(4), UNSIGNED	Reference Count
12	0C	BINARY(4), UNSIGNED	In-Use Indicator
16	10	BINARY(4), UNSIGNED	Offset to Simple Reference Types
20	14	BINARY(4), UNSIGNED	Length of Simple Reference Types

Offset		Type	Field
Dec	Hex		
24	18	BINARY(4), UNSIGNED	Offset to Extended Reference Types
28	1C	BINARY(4), UNSIGNED	Length of Extended Reference Types
32	20	BINARY(4), UNSIGNED	Offset to Job List
36	24	BINARY(4), UNSIGNED	Jobs Returned
40	28	BINARY(4), UNSIGNED	Jobs Available
Offset determined from *Offset to Simple Reference Types* field		Qp0l_Sim_Ref_Types_Output Structure	Simple Reference Types Structure See "Simple Object Reference Types Structure Description (*Qp0l_Sim_Ref_Types_Output*)" on page 389 for a description of this structure.
Offset determined from the *Offset to Extended Reference Types* field		Qp0l_Ext_Ref_Types_Output Structure	Extended Reference Types Structure. See "Extended Object Reference Types Structure Description (*Qp0l_Ext_Ref_Types_Output*)" on page 389 for a description of this structure. The reference counts contained within this structure represent the number of references for all jobs in the job list.
Offset determined from *Offset to Job List* field		Qp0l_Job_Using_Object Structure	Referencing job list. The "Job Using Object Structure Description (*Qp0l_Job_Using_Object*)" will be repeated for each job.

Job Using Object Structure Description (*Qp0l_Job_Using_Object*)

This structure is imbedded within the RORO0200 format. It is used to return information about a job that is known to be holding a reference on the object.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4), UNSIGNED	Displacement to Simple Reference Types
4	4	BINARY(4), UNSIGNED	Length of Simple Reference Types
8	8	BINARY(4), UNSIGNED	Displacement to Extended Reference Types
12	0C	BINARY(4), UNSIGNED	Length of Extended Reference Types
16	10	BINARY(4), UNSIGNED	Displacement to Next Job Entry
20	14	CHAR(10)	Job Name
30	1E	CHAR(10)	Job User
40	28	CHAR(6)	Job Number
46	2E	CHAR(2)	Reserved (Binary 0)
48	30	BINARY(4), UNSIGNED	Displacement to iSeries NetServer Session List
52	34	BINARY(4), UNSIGNED	iSeries NetServer Sessions Returned
Offset determined from the *Displacement to Simple Reference Types* field		Qp0l_Sim_Ref_Types_Output Structure	Simple Reference Types Structure. See "Simple Object Reference Types Structure Description (*Qp0l_Sim_Ref_Types_Output*)" on page 389 for a description of this structure.
Offset determined from the *Displacement to Extended Reference Types* field		Qp0l_Ext_Ref_Types_Output Structure	Extended Reference Types Structure. See "Extended Object Reference Types Structure Description (*Qp0l_Ext_Ref_Types_Output*)" on page 389 for a description of this structure. The reference counts contained within this structure represent the number of references for this specific job.

Offset		Type	Field
Dec	Hex		
Offset determined from the *Displacement to iSeries NetServer Session List* field		Qp0l_Session_Using_Object Structure	iSeries NetServer Session Using Object Structure. See iSeries NetServer Session Using Object Structure Description (page "iSeries NetServer Session Using Object Structure Description (*Qp0l_Session_Using_Object Structure*)" on page 391) for a description of this structure. The information within this structure represent the iSeries NetServer sessions which have a reference to the object.

Simple Object Reference Types Structure Description (*Qp0l_Sim_Ref_Types_Output*)

This structure is imbedded within the RORO0100 and RORO0200 formats. It is used to return object reference type information.

Each binary field reference type will be set to either 0 or a positive value that represents the number of references for that type. This number will have different meanings depending on the structure it is imbedded within. When this structure is imbedded within a RORO0100 output, or imbedded within the header portion of the RORO0200 output, then these values represent the number of known references of this type. When this structure is imbedded within a specific job list entry, then these values represent the number of references for that specific type within that specific job itself.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4), UNSIGNED	Read Only
4	4	BINARY(4), UNSIGNED	Write Only
8	8	BINARY(4), UNSIGNED	Read/Write
12	0C	BINARY(4), UNSIGNED	Execute
16	10	BINARY(4), UNSIGNED	Share with Readers Only
20	14	BINARY(4), UNSIGNED	Share with Writers Only
24	18	BINARY(4), UNSIGNED	Share with Readers and Writers
28	1C	BINARY(4), UNSIGNED	Share with neither Readers nor Writers
32	20	BINARY(4), UNSIGNED	Attribute Lock
36	24	BINARY(4), UNSIGNED	Save Lock
40	28	BINARY(4), UNSIGNED	Internal Save Lock
44	2C	BINARY(4), UNSIGNED	Link Changes Lock
48	30	BINARY(4), UNSIGNED	Checked Out
52	34	CHAR(10)	Checked Out User Name
62	3E	CHAR(2)	Reserved (Binary 0)

Extended Object Reference Types Structure Description (*Qp0l_Ext_Ref_Types_Output*)

This structure is imbedded within the RORO0200 format. It is used to return object reference type information.

Each binary field reference type will be set to either 0 or a positive value that represents the number of references for that type. This number will have different meanings depending on the structure it is imbedded within. When this structure is imbedded within the header portion of the RORO0200 output, then these values represent the number of jobs in the job list that contains a reference of this type. When this structure is imbedded within a specific job list entry, then these values represent the number of references for that specific type within that specific job itself.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4), UNSIGNED	Read Only, Share with Readers Only
4	4	BINARY(4), UNSIGNED	Read Only, Share with Writers Only
8	8	BINARY(4), UNSIGNED	Read Only, Share with Readers and Writers
12	0C	BINARY(4), UNSIGNED	Read Only, Share with neither Readers nor Writers
16	10	BINARY(4), UNSIGNED	Write Only, Share with Readers Only
20	14	BINARY(4), UNSIGNED	Write Only, Share with Writers Only
24	18	BINARY(4), UNSIGNED	Write Only, Share with Readers and Writers
28	1C	BINARY(4), UNSIGNED	Write Only, Share with neither Readers nor Writers
32	20	BINARY(4), UNSIGNED	Read/Write, Share with Readers Only
36	24	BINARY(4), UNSIGNED	Read/Write, Share with Writers Only
40	28	BINARY(4), UNSIGNED	Read/Write, Share with Readers and Writers
44	2C	BINARY(4), UNSIGNED	Read/Write, Share with neither Readers nor Writers
48	30	BINARY(4), UNSIGNED	Execute, Share with Readers Only
52	34	BINARY(4), UNSIGNED	Execute, Share with Writers Only
56	38	BINARY(4), UNSIGNED	Execute, Share with Readers and Writers
60	3C	BINARY(4), UNSIGNED	Execute, Share with neither Readers nor Writers
64	40	BINARY(4), UNSIGNED	Execute/Read, Share with Readers Only
68	44	BINARY(4), UNSIGNED	Execute/Read, Share with Writers Only
72	48	BINARY(4), UNSIGNED	Execute/Read, Share with Readers and Writers
76	4C	BINARY(4), UNSIGNED	Execute/Read, Share with neither Readers nor Writers
80	50	BINARY(4), UNSIGNED	Attribute Lock
84	54	BINARY(4), UNSIGNED	Save Lock
88	58	BINARY(4), UNSIGNED	Internal Save Lock
92	5C	BINARY(4), UNSIGNED	Link Changes Lock
96	60	BINARY(4), UNSIGNED	Current Directory
100	64	BINARY(4), UNSIGNED	Root Directory
104	68	BINARY(4), UNSIGNED	File Server Reference
108	6C	BINARY(4), UNSIGNED	File Server Working Directory
112	70	BINARY(4), UNSIGNED	Checked Out
116	74	CHAR(10)	Checked Out User Name
126	7E	CHAR(2)	Reserved (Binary 0)

iSeries NetServer Session Using Object Structure Description (*Qp0l_Session_Using_Object Structure*)

This structure is imbedded within the RORO0200 format. It is used to return information for sessions having a reference.

Note: iSeries NetServer refers to iSeries Support for Windows Network Neighborhood.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(8), UNSIGNED	Session identifier
8	8	BINARY(4), UNSIGNED	Displacement to Next iSeries NetServer Session Entry
12	0C	CHAR(10)	User Name
22	16	CHAR(15)	Workstation Name
37	25	CHAR(45)	Workstation Address
82	52	CHAR(14)	Reserved (Binary 0)

Field Descriptions for RORO0100 and RORO0200 Output Structures and their Imbedded Structures

Attribute Lock. Attribute changes are prevented.

Bytes Available. Number of bytes of output data that was available to be returned.

Bytes Returned. Number of bytes returned in the output buffer.

Checked Out. Indicates whether the object is currently checked out. If it is checked out, then the *Checked Out User Name* contains the name of the user who has it checked out.

Checked Out User Name. Contains the name of the user who has the object checked out, when the *Checked Out* field indicates that it is currently checked out. This field is set to blanks (x'40) if the object is not checked out.

Current Directory. The object is a directory that is being used as the current directory of the job.

Displacement to Extended Reference Types. Displacement from the beginning of the structure containing this field to the beginning of the Extended Reference Types structure. If this field is 0, then no extended reference types were available to be returned, or not enough space was provided to include any portion of the Extended Reference Types structure.

Displacement to iSeries NetServer Session List. Displacement from the beginning of the structure containing this field to the first iSeries NetServer Session Using Object structure. If this field is 0, then there are no sessions in the list. If the File Server Reference and the File Server Working Directory fields are set to 0, then this field will be set to 0.

Displacement to Next iSeries NetServer Session Entry. Displacement from the beginning of the structure containing this field to the beginning of the next iSeries NetServer Session Using Object structure. If this field is 0, then there are no more sessions in the list.

Displacement to Next Job Entry. Displacement from the beginning of the structure containing this field to the beginning of the next Job Using Object structure. If this field is 0, then there are no more jobs in the list, or not enough space was provided to include any more Job Using Object structures.

Displacement to Simple Reference Types. Displacement from the beginning of the structure containing this field to the beginning of the Simple Reference Type structure. If this field is 0, then no simple reference types were available to be returned, or not enough space was provided to include any portion of the Simple Reference Types structure.

Execute. Execute only access.

Execute, Share with Readers Only. Execute only access. The sharing mode allows sharing with read and execute access intents only.

Execute, Share with Readers and Writers. Execute only access. The sharing mode allows sharing with read, execute, and write access intents.

Execute, Share with Writers Only. Execute only access. The sharing mode allows sharing with write access intents only.

Execute, Share with neither Readers nor Writers. Execute only access. The sharing mode allows sharing with no other access intents.

Execute/Read, Share with Readers Only. Execute and read access. The sharing mode allows sharing with read and execute access intents only.

Execute/Read, Share with Readers and Writers. Execute and read access. The sharing mode allows sharing with read, execute, and write access intents.

Execute/Read, Share with Writers Only. Execute and read access. The sharing mode allows sharing with write access intents only.

Execute/Read, Share with neither Readers nor Writers. Execute and read access. The sharing mode allows sharing with no other access intents.

Extended Reference Types Structure. This is a Qp0l_Ext_Ref_Types_Output structure containing fields that indicate different types of references that may be held on an object. Some of these are actually a grouping of multiple **Simple Reference Types** that were known to have been specified by the referring instance. These are not additional references; they are a redefinition of the same references described in the Simple Reference Types structure.

File Server Reference. The File Server is holding a generic reference on the object on behalf of a client.

» If this field is not 0, then iSeries NetServer session information may have been returned. «

File Server Working Directory. The object is a directory, and the File Server is holding a working directory reference on it on behalf of a client. » If this field is not 0, then iSeries NetServer session information may have been returned. «

In-Use Indicator The object is currently in-use. NOTE: This indicator will be set to one of the following values:

QP0L_OBJECT_NOT_IN_USE (0)
: The object is not in use and all of the reference type fields returned are 0.

QP0L_OBJECT_IN_USE (1)
: The object is in use. At least one of the reference type fields is greater than 0. This condition may occur even if the Reference Count field's value is 0.

Internal Save Lock. The object is being referenced internally during a save operation on a different object.

iSeries NetServer Sessions Returned. The number of iSeries NetServer Session Using Object structures returned for the job.

Job Name. Name of the job.

Job Number. Number associated with the job.

Job User. User profile associated with the job.

Jobs Available. Number of referencing jobs available. This may be greater than the **Jobs Returned** field when the caller did not provide enough space to receive all of the job information.

Jobs Returned. Number of referencing jobs returned in the job list.

Length of Extended Reference Types. Length of the Extended Reference Types information.

Length of Simple Reference Types. Length of the Simple Reference Types information.

Link Changes Lock. Changes to links in the directory are prevented.

Offset to Extended Reference Types. Offset from the beginning of the *Receiver_Ptr* to the beginning of the Extended Reference Types structure. If this field is 0, then no extended reference types were available to be returned, or not enough space was provided to include any portion of the Extended Reference Types structure.

Offset to Job List. Offset from the beginning of the *Receiver_Ptr* to the beginning of the first Job Using Object structure. If this field is 0, then there are no jobs in the list.

Offset to Simple Reference Types. Offset from the beginning of the *Receiver_Ptr* to the beginning of the Simple Reference Type structure. If this field is 0, then no simple reference types were available to be returned, or not enough space was provided to include any portion of the Simple Reference Types structure.

Read Only. Read only access.

Read Only, Share with Readers Only. Read only access. The sharing mode allows sharing with read and execute access intents only.

Read Only, Share with Readers and Writers. Read only access. The sharing mode allows sharing with read, execute, and write access intents.

Read Only, Share with Writers Only. Read only access. The sharing mode allows sharing with write access intents only.

Read Only, Share with neither Readers nor Writers. Read only access. The sharing mode allows sharing with no other access intents.

Read/Write. Read and write access.

Read/Write, Share with Readers Only. Read and write access. The sharing mode allows sharing with read and execute access intents only.

Read/Write, Share with Readers and Writers. Read and write access. The sharing mode allows sharing with read, execute, and write access intents.

Read/Write, Share with Writers Only. Read and write access. The sharing mode allows sharing with write access intents only.

Read/Write, Share with neither Readers nor Writers. Read and write access. The sharing mode allows sharing with no other access intents.

Reference Count. Current number of references on the object. NOTE: This may be 0 even though the In-Use Indicator indicates that the object is in use.

Referencing Job List. Variable length list of Qp0l_Job_Using_Object structures for jobs that are currently referencing the object.

Root Directory. The object is a directory that is being used as the root directory of the job.

Save Lock. The object is being referenced by an object save operation.

Session identifier. Unique identifier for the iSeries NetServer session.

Share with Readers Only. The sharing mode allows sharing with read and execute access intents only.

Share with Readers and Writers. The sharing mode allows sharing with read, execute, and write access intents.

Share with Writers Only. The sharing mode allows sharing with write access intents only.

Share with neither Readers nor Writers. The sharing mode allows sharing with no other access intents.

Simple Reference Types Structure. This is a Qp0l_Sim_Ref_Types_Output structure containing fields that indicate different types of references that may be held on an object.

User name. The name of the user that is associated with the iSeries NetServer session.

Workstation address. The IP address of the workstation from which the iSeries NetServer session to the server was established. If this information is not available, this field will be set to blanks.

Workstation name. The name of the workstation from which the iSeries NetServer session to the server was established. If this information is not available, this field will be set to blanks.

Write Only. Write only access.

Write Only, Share with Readers Only. Write only access. The sharing mode allows sharing with read and execute access intents only.

Write Only, Share with Readers and Writers. Write only access. The sharing mode allows sharing with read, execute, and write access intents.

Write Only, Share with Writers Only. Write only access. The sharing mode allows sharing with write access intents only.

Write Only, Share with neither Readers nor Writers. Write only access. The sharing mode allows sharing with no other access intents.

Error Messages

CPF3C21 E	Format name &1 is not valid.
CPF3C24 E	Length of the receiver variable is not valid.

CPF3C36 E	Number of parameters, &1, entered for this API was not valid.
CPF3CF1 E	Error code parameter not valid.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. Since both available formats are variable length, following are the recommended minimum lengths pertaining to their corresponding formats:
   - RORO0100: The size of a RORO0100 Output structure plus the size of a *Simple Reference Types* structure.
   - RORO0200: This structure varies dynamically, and therefore there is no formula that can yield a size large enough to always retrieve all of the available information. However, programs may consider first calling QP0LROR with the RORO0100 format. This will quickly return the number of references currently on the object. Then the program could allocate a buffer equal in size to: size of a *Job Using Object* structure (including the size of » the Simple and Extended Reference Type structures and the iSeries NetServer Session Using Object structure«) multiplied by the number of references, and then add the sizes of a RORO0100 output, RORO0200 output, and Simple Reference Types structures. Now the program could call QP0LROR with the RORO0200 format requested and the computed size.

     If the RORO0200 format was specified, but there was not enough space provided to receive a complete list of job information, then only those job entries that completely fit in the buffer will be returned. The RORO0200 output structure contains a field called *JobsAvailable* that will always contain the total number of referencing jobs that were available for returning to the caller at that instance in time.

   Notes

   - There are no locks obtained on the object while this API is running. Therefore, when this API is used on an object that is actively in use (e.g., its lock and reference state is changing while this API is running), some fields in the returned information may be inconsistent with other fields returned on the same invocation of QP0LROR.
   - The number of references on the object may change between multiple calls to this API. Therefore, the above formula for calculating output buffer size for a RORO0200 format may not be enough space under all conditions.
   - There are some reference types that are obtained on the object without incrementing the object's reference count. This could result in a reference count of zero while the object contains reference types. In this instance, the above formula for calculating output buffer size for a RORO0200 format may not be enough space.
2. The list of simple object reference types in the base portions of the RORO0100 and RORO0200 output structures may not contain complete information for objects residing in file systems other than the "root" (/), QOpenSys, and user-defined file systems. The simple reference types will, however, be set in the job array elements in the RORO0200 output structure for any file system.
3. The list of object reference types in the RORO0200 output formats may be an incomplete list of references for objects residing in file systems other than the "root" (/), QOpenSys, and user-defined file systems. Objects in some of the other file systems can be locked with interfaces that do not use the integrated file system. Therefore, references returned by this API will only be references that were obtained as part of an integrated file system operation, or an operation that cause the integrated file system operation to occur.
4. Under some circumstances, the list of jobs that are referencing the object may be incomplete. However, jobs not listed in the job list may still have their references listed in the RORO0100 output. This occurs when system programs obtain references directly on an object without obtaining an open descriptor for the object.

5. At some instances during the save or restore of an integrated file system object, the object may have references held by the job even though its reference count is 0.
6. The Network File System (NFS) will only be returning references that are locally obtained on the object. Any references that the remote system may have on the remote object are not returned by this API.
7. Use of this API on an object accessed via the QFileSvr.400 file system will not return any job references, even if the object was opened using the QFileSvr.400 client.
8. This type of reference information is also viewable through the iSeries Navigator application. The terminology, however, differs in that iSeries Navigator refers to this type of information as ″Usage″ information instead of ″Reference″ information.

Related Information

- The <**qp0lror.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- Retrieve Referenced Objects (QP0LRRO) API

Example

See Code disclaimer information for information pertaining to code examples.

The following is an example use of this API.

```
#include <qp0lror.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void main()
{
    struct PathNameStruct
    {
 Qlg_Path_Name_T header;
 char p[50];
    };

    struct PathNameStruct path;

    char pathName[] = "/CustomerData";

    Qus_EC_t errorCode;

    /* Define a constant for the number of output buffer bytes
       provided for the RORO0100 format.                    */
#define OUTPUT_BYTES_RORO0100                                \
             (sizeof(Qp0l_RORO0100_Output_T) +               \
              sizeof(Qp0l_Sim_Ref_Types_Output_T) +   \
              100)       /* Pad space for potential gap between
                            the 2 structures.               */

    /* Declare some space for the RORO0100 output.           */
    char output100Buf[OUTPUT_BYTES_RORO0100];

    /* Declare a pointer for retrieving the RORO0100 format.  */
    Qp0l_RORO0100_Output_T *output100P;

    /* Declare a pointer to retrieve the RORO0200 format.     */
    Qp0l_RORO0200_Output_T *output200P;

    /* Declare a job using object pointer.                    */
    Qp0l_Job_Using_Object_T *jobP;

    unsigned outputBufSize;

    /* Set output buffer pointer and length for retrieving the
```

```
      RORO0100 format.                                    */
   output100P = (Qp0l_RORO0100_Output_T *)output100Buf;

   /* Setup the object's path name structure.             */
   memset(&path, 0, sizeof(path));
   path.header.CCSID = 37;
   memcpy(path.header.Country_ID,"US",2);
   memcpy(path.header.Language_ID,"ENU",3);
   path.header.Path_Type = QLG_CHAR_SINGLE;
   path.header.Path_Length = strlen(pathName);
   path.header.Path_Name_Delimiter[0] = '/';
   memcpy(path.p, pathName, path.header.Path_Length);

   /* Setup the error code structure to cause the error to be
      returned within the error structure.                */
   errorCode.Bytes_Provided = sizeof(errorCode);
   errorCode.Bytes_Available = 0;

   /* First call QP0LROR to get the short format.  We will
      use that information about references to conditionally
      allocate more space and then get the longer
      running format's information.                       */
   QP0LROR(output100P,
    OUTPUT_BYTES_RORO0100,
    QP0LROR_RORO0100_FORMAT,
    (Qlg_Path_Name_T *) &path,
    &errorCode);

   /* Check if an error occurred.                         */
   if (errorCode.Bytes_Available != 0)
   {
printf("Error occurred for RORO0100.\n");
return;
   }

   /* Check if we received any references that might be
      associated with a job.  If not, return.             */
   if (output100P->Count == 0)
   {
printf("QP0LROR returned a reference count of %d\n",
       output100P->Count);
return;
   }

   /* If we get here, then we have at least 1 reference that
      may be identifiable to a job.  We will call the
      QP0LROR API to get the RORO0200 format.  First we
      compute a buffer size to use.  Note:  this calculation
      sums up the sizes of all structures contained within
      the RORO0200 format, but doesn't consider gaps between
      each of the structure.  To attempt to cover potential
      gaps between structures, an extra 1000 bytes is being
      allocated and room for 10 additional jobs.          */
   outputBufSize =
     sizeof(Qp0l_RORO0200_Output_T) +
     sizeof(Qp0l_Sim_Ref_Types_Output_T) +
     sizeof(Qp0l_Ext_Ref_Types_Output_T) +
     ((output100P->Count + 10) *
      (sizeof(Qp0l_Job_Using_Object_T) +
sizeof(Qp0l_Sim_Ref_Types_Output_T) +
sizeof(Qp0l_Ext_Ref_Types_Output_T)
) + 1000
      );

   if (NULL == (output200P =
  (Qp0l_RORO0200_Output_T *)malloc(outputBufSize)))
   {
```

```
printf("No space available.\n");
return;
   }

   /* Retrieve object references.                             */
   QP0LROR(output200P,
    outputBufSize,
    QP0LROR_RORO0200_FORMAT,
    (Qlg_Path_Name_T *) &path,
    &errorCode);

   /* Check if an error occurred.                             */
   if (errorCode.Bytes_Available != 0)
   {
free(output200P);
printf("Error occurred for RORO0200.\n");
return;
   }

   /* If there was more information available than we had
      provided receiver space for, then we will allocate a
      larger buffer and try once again.  This could potentially
      keep reoccurring, but this example will stop after this
      second retry.                                           */
   if (output200P->BytesReturned < output200P->BytesAvailable)
   {
       /* Use the bytes available value to determine how much
          more buffer size is needed.  We will pad it with an
          extra 1000 bytes to try and handle more jobs obtaining
          references between calls to QP0LROR.               */
       outputBufSize = output200P->BytesAvailable + 1000;

if (NULL == (output200P = (Qp0l_RORO0200_Output_T *)
      realloc((void *)output200P,
       outputBufSize)))
{
    printf("No space available.\n");
    return;
}

QP0LROR(output200P,
 outputBufSize,
 QP0LROR_RORO0200_FORMAT,
 (Qlg_Path_Name_T *) &path,
 &errorCode);

       /* Check if an error occurred.                         */
if (errorCode.Bytes_Available != 0)
{
    free(output200P);
    printf("Error occurred for RORO0200 (2nd call).\n");
    return;
}
   }

   /* Print some output.                                      */
   printf("Reference count:  %d\n",output200P->Count);
   printf("Jobs returned:    %d\n",output200P->JobsReturned);

   if (output200P->JobsReturned > 0)
   {
jobP = (Qp0l_Job_Using_Object_T *)
  ((char *)output200P + output200P->JobsOffset);
printf("First job's name: %10.10s %10.10s %6.6s",
       jobP->Name,
       jobP->User,
       jobP->Number);
```

```
    }

    free(output200P);

    return;
}
```

Example Output:

```
Reference count:  1
Jobs returned:    1
First job's name: JOBNAME123 JOBUSER123 123456
```

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

Qp0lSaveStgFree()—Save Storage Free

Syntax

```
#include <Qp0lstdi.h>

int Qp0lSaveStgFree(
  Qlg_Path_Name_T         *Path_Name,
  Qp0l_StgFree_Function_t *UserFunction_ptr,
  void                    *Function_CtlBlk_ptr);
```

Service Program Name: QP0LLIB3
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 402.

The **Qp0lSaveStgFree()** function calls a user-supplied exit program to save i5/OS objects of type *STMF and, upon successful completion of the exit program, frees the storage for the object and marks the object as storage freed. The *STMF object and its attributes remain on the system, but the storage occupied by the *STMF object's data is deleted. The *STMF object cannot be used until it is restored to the system. This is accomplished by either of the following:

- Restoring the object using the RST command.
- Requesting an operation on the object, requiring one of the following, which will dynamically retrieve (restore) the *STMF object:
  - Accessing the object's data (**open()**, **creat()**, MOV, CPY, CPYFRMSTMF, or CPYTOSTMF).
  - Adding a new name to the object (RNM, ADDLNK, **link()**, **rename()**, **Qp0lRenameKeep()**, or **Qp0lRenameUnlink()**).
  - Checking out the object (CHKOUT).

The restore operation is done by calling a user-provided exit program registered against the Storage Extension exit point QIBM_QTA_STOR_EX400. For information on this exit point, see the Storage Extension Exit Program.

Qp0lSaveStgFree() returns EOFFLINE for an object that is already storage freed or returns EBUSY for an object that is checked out.

The user exit program can be either a procedure or a program.

Parameters

Path_Name

(Input) A pointer to a path name whose last component is the object that is saved and whose storage is freed. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path name format.

If the last component of the path name supplied on the call to **Qp0lSaveStgFree()** is a symbolic link, then **Qp0lSaveStgFree()** resolves and follows the link to its target and performs its normal **Qp0lSaveStgFree()** functions on that target. If the symbolic link refers to an object in a remote file system, **Qp0lSaveStgFree()** returns ENOTSUP to the calling program.

UserFunction_ptr

(Input) A pointer to a structure that contains information about the user exit program that the caller wants **Qp0lSaveStgFree()** to call to save an *STMF object. This user exit program can be either a procedure or a program. If this pointer is NULL, **Qp0lSaveStgFree()** does not call an exit program to save the object but does free the object's storage and marks it as storage freed.

User Function Pointer			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Function type flag
14	E	CHAR(10)	Program library
4	4	CHAR(10)	Program name
24	18	CHAR(1)	Multithreaded job action
25	19	CHAR(7)	Reserved
32	20	PP(*)	Procedure pointer to exit procedure

Function type flag. A flag that indicates whether the Save Storage Free exit program called by **Qp0lSaveStgFree()** is a procedure or a program. If the exit program is a procedure, this flag is set to 0, and the procedure pointer to exit procedure field points to the procedure called by **Qp0lSaveStgFree()**. If the exit program is a program, this flag is set to 1 and a program name and program library are provided, respectively, in the program name and program library fields. Valid values follow:

0 QP0L_USER_FUNCTION_PTR: A user procedure is called.
1 QP0L_USER_FUNCTION_PGM: A user program is called.

Multithreaded job action. (Input) A CHAR(1) value that indicates the action to take in a multithreaded job. The default value is QP0L_MLTTHDACN_SYSVAL. For release compatibility and for processing this parameter against the QMLTTHDACN system value, x'00, x'01', x'02', & x'03' are treated as x'F0', x'F1', x'F2', and x'F3'.

x'00' QP0L_MLTTHDACN_SYSVAL: The API evaluates the QMLTTHDACN system value to determine the action to take in a multithreaded job. Valid QMLTTHDACN system values follow:

- *'1'* Call the exit program. Do not send an informational message.
- *'2'* Call the exit program and send informational message CPI3C80.
- *'3'* The exit program is not called when the API determines that it is running in a multithreaded job. ENOTSAFE is returned.

x'01' QP0L_MLTTHDACN_NOMSG: Call the exit program. Do not send an informational message.
x'02' QP0L_MLTTHDACN_MSG: Call the exit program and send informational message CPI3C80.

x'03' QP0L_MLTTHDACN_NO: The exit program is not called when the API determines that it is running in a multithreaded job. ENOTSAFE is returned.

Procedure pointer to exit procedure. If the function type flag is 0, which indicates that a procedure is called instead of a program, this field contains a procedure pointer to the procedure that **Qp0lSaveStgFree()** calls. This field must be NULL if the function type flag is 1.

Program library. If the function type flag is 1, indicating a program is called, this field contains the library in which the program being called (identified by the program name field) is located. This field must be blank if the function type flag is 0.

Program name. If the function type flag is 1, indicating a program is called, this field contains the name of the program that is called. The program should be located in the library identified by the program library field. This field must be blank if the function type flag is 0.

Reserved. A reserved field. This field must be set to binary zero.

Function_CtlBlk_ptr
(Input) A pointer to any data that the caller of **Qp0lSaveStgFree()** wants to have passed to the user-defined Save Storage Free exit program that **Qp0lSaveStgFree()** calls to save an *STMF object. **Qp0lSaveStgFree()** does not process the data that is referred to by this pointer. The API passes this pointer as a parameter to the user-defined Save Storage Free exit program that was specified on its call. This is a means for the caller of **Qp0lSaveStgFree()** to pass information to and from the Save Storage Free exit program.

Authorities

The following table shows the authorization required for the Qp0lSaveStgFree() API.

Object Referred to	Authority Required	errno
Each directory, preceding the last component, in a *path name*	*RX	EACCES
Object	*SAVSYS or *RW	EACCES
Any called program pointed to by the *UserFunction_ptr* parameter	*X	EACCES
Any library containing the called program pointed to by the *UserFunction_ptr* parameter	*X	EACCES

Return Value

0 **Qp0lSaveStgFree()** was successful.
-1 **Qp0lSaveStgFree()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **Qp0lSaveStgFree()** is not successful, *errno* indicates one of the following errors:

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADNAME (page 540)]	

Error condition	Additional information
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EISDIR (page 544)]	
[ELOOP (page 544)]	
[EMFILE (page 543)]	
[ENAMETOOLONG (page 544)]	
[ENFILE (page 543)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOSPC (page 541)]	
[ENOSYSRSC (page 545)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EOFFLINE (page 545)]	
[EUNKNOWN (page 544)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPI3C80 I	An exit program has been called for which the threadsafety status was not known.
CPFA0D4 E	File system error occurred.
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

- This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
  - Where multiple threads exist in the job.
  - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
    - "Root" (/)
    - QOpenSys
    - User-defined
    - QNTC
    - QSYS.LIB
    - QOPT
    - Network File System
    - QFileSvr.400

- If the Save Storage Free exit program calls the SAV command or the **QsrSave** function or any other function that is not threadsafe, and there are secondary threads active in the job, **Qp0lSaveStgFree()** may fail as a result.
- If the Save Storage Free exit program is not threadsafe or uses a function that is not threadsafe, then **Qp0lSaveStgFree()** is not threadsafe.
- This function will fail with error code [EINVAL] if the stream file this function is operating on is a virtual volume.

Related Information

- The <**Qp0lstdi.h**> file
- "QlgSaveStgFree()—Save Storage Free (using NLS-enabled path name)" on page 292—Save Storage Free (using NLS-enabled path name)
- "Save Storage Free Exit Program" on page 535

Example

See Code disclaimer information for information pertaining to code examples.

See **"Qp0lGetAttr()—Get Attributes" on page 326** description for a code example that shows a call to **Qp0lSaveStgFree()** by using a procedure as the exit program. This API also shows an example of a call to **Qp0lGetAttr()**.

API introduced: V4R3

Top | Backup and Recovery APIs | UNIX-Type APIs | APIs by category

---

Qp0lSetAttr()—Set Attributes

Syntax

```
#include <Qp0lstdi.h>
int Qp0lSetAttr
       (Qlg_Path_Name_T     *Path_Name,
        char                *Buffer_ptr,
        uint                 Buffer_Size,
        uint                 Follow_Symlnk, ...);
```

Service Program Name: QP0LLIB3
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 412.

The **Qp0lSetAttr()** function sets one of a set of defined attributes, on each call, for the object that is referred to by the input **Path_Name*. The object must exist, the user must have authority to it, and the attribute must be supported by the file system to which the object belongs. When an attribute is not supported by the file system, **Qp0lSetAttr()** will fail with ENOTSUP. See the "Usage Notes" on page 412 for more information.

If the last component of the *Path_Name* parameter is a symbolic link, the **Qp0lSetAttr()** either sets the attribute of the symbolic link or sets the attribute of the object that the symbolic link names. This depends on the value of the *Follow_Symlnk* parameter.

All times that are set by **Qp0lSetAttr()** are in seconds since the Epoch so that they are consistent with UNIX-type APIs. The Epoch is the time 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time. If the system date is set prior to 1970, all time values will be zero.

Parameters

Path_Name

(Input) The path name of the object for which attribute information is set. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path name format.

Buffer_ptr

(Input) A pointer to a buffer containing a constant that identifies the attribute and the value for the attribute that **Qp0lSetAttr()** sets. The number of bytes allocated for this buffer is in the *Buffer_Size* parameter.

The following table describes the format of the entry in the buffer.

Buffer Pointer			
Offset			
Dec	**Hex**	**Type**	**Field**
0	0	BINARY(4)	Offset to next attribute entry
4	4	BINARY(4)	Attribute identification
8	8	BINARY(4)	Size of attribute data
12	C	CHAR(4)	Reserved
16	10	CHAR(*)	Attribute data

Attribute data. The value to which the attribute is set.

Attribute identification. The constant identifying the attribute being set. Valid values are:

4 QP0L_ATTR_CREATE_TIME: (UNSIGNED (BINARY(4)) The time the object was created.

5 QP0L_ATTR_ACCESS_TIME: (UNSIGNED (BINARY(4)) The time the object's data was last accessed.

7 QP0L_ATTR_MODIFY_TIME: (UNSIGNED (BINARY(4)) The time the object's data was last changed.

17 QP0L_ATTR_PC_READ_ONLY: (CHAR(1)) Whether the object can be written to or deleted, have its extended attributes changed or deleted, or have its size changed. Valid values are:

x'00' QP0L_PC_NOT_READONLY: The object can be changed.

x'01' QP0L_PC_READONLY: The object cannot be changed.

18 QP0L_ATTR_PC_HIDDEN: (CHAR(1)) Whether the object can be displayed using an ordinary directory listing.

x'00' QP0L_PC_NOT_HIDDEN: The object is not hidden.

x'01' QP0L_PC_HIDDEN: The object is hidden.

19 QP0L_ATTR_PC_SYSTEM: (CHAR(1)) Whether the object is a system file and is excluded from normal directory searches.

x'00' QP0L_PC_NOT_SYSTEM: The object is not a system file.

x'01' QP0L_PC_SYSTEM: The object is a system file.

20 QP0L_ATTR_PC_ARCHIVE: (CHAR(1)) Whether the object has changed since the last time the file was saved or reset by a PC client.

x'00' QP0L_PC_NOT_CHANGED: The object has not changed.

x'01' QP0L_PC_CHANGED: The object has changed.

21 QP0L_ATTR_SYSTEM_ARCHIVE: (CHAR(1)) Whether the object has changed and needs to be saved. It is set on when an object's change time is updated, and set off when the object has been saved.

x'00' QP0L_SYSTEM_NOT_CHANGED: The object has not changed and does not need to be saved.

x'01' QP0L_SYSTEM_CHANGED: The object has changed and does need to be saved.

22 QP0L_ATTR_CODEPAGE: (BINARY(4)) The code page used to derive a coded character set identifier (CCSID) used for the data in the file or the extended attributes of the directory.

26 QP0L_ATTR_ALWCKPWRT: (CHAR(1)) Whether a stream file (*STMF) can be shared with readers and writers during the save-while-active checkpoint processing. Setting this attribute may cause unexpected results. See the Back up your server topic for details on this attribute.

x'00' QP0L_NOT_ALWCKPWRT: The object can be shared with readers only.

x'01' QP0L_ALWCKPWRT: The object can be shared with readers and writers.

27 QP0L_ATTR_CCSID: (BINARY(4)) The CCSID of the data and extended attributes of the object.

31 QP0L_ATTR_DISK_STG_OPT (CHAR(1)) Which option should be used to determine how auxiliary storage is allocated by the system for the specified object. The option will take effect immediately and be part of the next auxiliary storage allocation for the object. This option can only be specified for byte stream files in the "root" (/), QOpenSys and user-defined file systems. This option will be ignored for *TYPE1 byte stream files. Valid values are:

x'00' QP0L_STG_NORMAL: The auxiliary storage will be allocated normally. That is, as additional auxiliary storage is required, it will be allocated in logically sized extents to accomodate the current space requirement, and anticipated future requirements, while minimizing the number of disk I/O operations. If the QP0L_ATTR_DISK_STG_OPT attribute has not been specified for an object, this value is the default.

x'01' QP0L_STG_MINIMIZE: The auxiliary storage will be allocated to minimize the space used by the object. That is, as additional auxiliary storage is required, it will be allocated in small sized extents to accomodate the current space requirement. Accessing an object composed of many small extents may increase the number of disk I/O operations for that object.

x'02' QP0L_STG_DYNAMIC: The system will dynamically determine the optimum auxiliary storage allocation for the object, balancing space used versus disk I/O operations. For example, if a file has many small extents, yet is frequently being read and written, then future auxiliary storage allocations will be larger extents to minimize the number of disk I/O operations. Or, if a file is frequently truncated, then future auxiliary storage allocations will be small extents to minimize the space used. Additionally, information will be maintained on the byte stream file sizes for this system and its activity. This file size information will also be used to help determine the optimum auxiliary storage allocations for this object as it relates to the other objects sizes.

32 QP0L_ATTR_MAIN_STG_OPT: (CHAR(1)) Which option should be used to determine how main storage is allocated and used by the system for the specified object. The option will take effect the next time the specified object is opened. This option can only be specified for byte stream files in the "root" (/), QOpenSys and user-defined file systems. Valid values are:

x'00' QP0L_STG_NORMAL: The main storage will be allocated normally. That is, as much main storage as possible will be allocated and used. This minimizes the number of disk I/O operations since the information is cached in main storage. If the QP0L_ATTR_MAIN_STG_OPT attribute has not been specified for an object, this value is the default.

x'01' QP0L_STG_MINIMIZE: The main storage will be allocated to minimize the space used by the object. That is, as little main storage as possible will be allocated and used. This minimizes main storage usage while increasing the number of disk I/O operations since less information is cached in main storage.

x'02' QP0L_STG_DYNAMIC: The system will dynamically determine the optimum main storage allocation for the object depending on other system activity and main storage contention. That is, when there is little main storage contention, as much storage as possible will be allocated and used to minimize the number of disk I/O operations. And when there is significant main storage contention, less main storage will be allocated and used to minimize the main storage contention. » This option only has an effect when the storage pool's paging option is *CALC. When the storage pool's paging option is *FIXED, the behavior is the same as *NORMAL. When the object is accessed through a file server, this option has no effect. Instead, its behavior is the same as *NORMAL. «

35 QP0L_ATTR_CRTOBJSCAN: (CHAR(1)) Whether the objects created in a directory will be scanned when exit programs are registered with any of the integrated file system scan-related exit points.

The integrated file system scan-related exit points are:

- "Integrated File System Scan on Close Exit Program" on page 513
- "Integrated File System Scan on Open Exit Program" on page 523.

This attribute can only be specified for directories in the "root" (/), QOpenSys and user-defined file systems. Even though this attribute can be set for *TYPE1 and *TYPE2 directories, only objects which are in *TYPE2 directories will actually be scanned, no matter what value is set for this attribute.

Valid values are:

x'00' QP0L_SCANNING_NO: After an object is created in the directory, the object will not be scanned according to the rules described in the scan-related exit programs.

Note: If the Scan file systems control (QSCANFSCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore.

x'01' QP0L_SCANNING_YES: After an object is created in the directory, the object will be scanned according to the rules described in the scan-related exit programs if the object has been modified or if the scanning software has been updated since the last time the object was scanned. If the QP0L_ATTR_CRTOBJSCAN attribute has not been specified for a directory, this value is the default.

x'02' QP0L_SCANNING_CHGONLY: After an object is created in the directory, the object will be scanned according to the rules described in the scan-related exit programs only if the object has been modified since the last time the object was scanned. It will not be scanned if the scanning software has been updated. This attribute only takes effect if the Scan file systems control (QSCANFSCTL) system value has *USEOCOATR specified. Otherwise, it will be treated as if the attribute is QP0L_SCANNING_YES.

Note: If the Scan file systems control (QSCANFSCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore.

36 QP0L_ATTR_SCAN: (CHAR(1)) Whether the object will be scanned when exit programs are registered with any of the integrated file system scan-related exit points.

The integrated file system scan-related exit points are:

- "Integrated File System Scan on Close Exit Program" on page 513
- "Integrated File System Scan on Open Exit Program" on page 523.

This attribute can only be specified for stream files in the "root" (/), QOpenSys and user-defined file systems that are not virtual volumes or network server storage spaces. Even though this attribute can be set for objects in *TYPE1 and *TYPE2 directories, only objects which are in *TYPE2 directories will actually be scanned, no matter what value is set for this attribute.

Valid values are:

x'00' QP0L_SCANNING_NO: The object will not be scanned according to the rules described in the scan-related exit programs.

Note: If the Scan file systems control (QSCANFSCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore.

x'01' QP0L_SCANNING_YES: The object will be scanned according to the rules described in the scan-related exit programs if the object has been modified or if the scanning software has been updated since the last time the object was scanned. If the QP0L_ATTR_SCAN attribute has not been specified for an object, this value is the default.

x'02' QP0L_SCANNING_CHGONLY: The object will be scanned according to the rules described in the scan-related exit programs only if the object has been modified since the last time the object was scanned. It will not be scanned if the scanning software has been updated. This attribute only takes effect if the Scan file systems control (QSCANFSCTL) system value has *USEOCOATR specified. Otherwise, it will be treated as if the attribute is QP0L_SCANNING_YES.

Note: If the Scan file systems control (QSCANFSCTL) value *NOPOSTRST is not specified when an object with this attribute is restored, the object will be scanned at least once after the restore.

38 QP0L_ATTR_ALWSAV: (CHAR(1)) Whether the object can be saved or not.

Note: It is highly recommended that this attribute not be changed for any system created objects.

Valid values are:

x'00' QP0L_ALWSAV_NO: This object will not be saved when using the Save Object (SAV) command or the QsrSave() API.

Additionally, if this object is a directory, none of the objects in the directory's subtree will be saved unless they were explicitly specified as an object to be saved. The subtree includes all subdirectories and the objects within those subdirectories.

Note: If this attribute is chosen for an object that has private authorities associated with it, or is chosen for the directory of an object that has private authorities associated with it, then the following consideration applies. When the private authorities are saved, the fact that an object has the QP0L_ALWSAV_NO attribute is not taken into consideration. (Private authorities can be saved using either the Save System (SAVSYS) or Save Security Data (SAVSECDTA) command or the Save Object List (QSRSAVO) API.) Therefore, when a private authority is restored using the Restore Authority (RSTAUT) command, message CPD3776 will be seen for each object that was not saved either because it had the QP0L_ALWSAV_NO attribute specified, or because the object was not specified on the save and it was in a directory that had the QP0L_ALWSAV_NO attribute specified.

x'01' QP0L_ALWSAV_YES: This object will be saved when using the Save Object (SAV) command or the QsrSave() API. If the QP0L_ATTR_ALWSAV attribute has not been specified for an object, this value is the default.

39 QP0L_ATTR_RSTDRNMUNL: (CHAR(1)) Restricted renames and unlinks for objects within a directory. Objects can be linked into a directory that has this attribute set on, but cannot be renamed or unlinked from it unless one or more of the following are true for the user performing the operation:

- The user is the owner of the object.
- The user is the owner of the directory.
- The user has *ALLOBJ special authority.

This restriction only applies to directories. Other types of object can have this attribute on, however, it will be ignored. In addition, this attribute can only be specified for objects within the Network File System (NFS), QFileSvr.400, "root" (/), QOpenSys, or user-defined file systems. Both the NFS and QFileSvr.400 file systems support this attribute by passing it to the server and surfacing it to the caller. This attribute is also equivalent to the S_ISVTX mode bit for an object. Valid values are:

x'00' QP0L_RSTDRNMUNL_OFF: No additional restrictions for rename and unlink operations.

x'01' QP0L_RSTDRNMUNL_ON: Additional restrictions for rename and unlink operations.

41 QP0L_ATTR_CRTOBJAUD: (CHAR(10)) The create object auditing value associated with the directory. This is the auditing value given to any objects created in the directory. This attribute can only be specified for directories in the "root" (/), QOpenSys, QSYS.LIB, independent ASP QSYS.LIB, QFileSvr.400 and user-defined file systems.

Valid values are:

**SYSVAL*
QP0L_AUD_SYSVAL: The object auditing value for the objects created in the directory is determined by the system auditing value (QCRTOBJAUD).

**NONE* QP0L_AUD_NONE: No auditing occurs for this object when it is read or changed regardless of the user who is accessing the object.

**USRPRF*
QP0L_AUD_USRPRF: Audit this object only if the current user is being audited. The current user is tested to determine if auditing should be done for this object. The user profile can specify if only change access is audited or if both read and change accesses are audited for this object. The OBJAUD parameter of the Change User Auditing (CHGUSRAUD) command is used to change the auditing for a specific user.

**CHANGE*
QP0L_AUD_CHANGE: Audit all change access to this object by all users on the system.

**ALL* QP0L_AUD_ALL: Audit all access to this object by all users on the system. All access is defined as a read or change operation.

200 QP0L_ATTR_RESET_DATE: (UNSIGNED (BINARY(2)) The count of the number of days an object has been used. Usage has different meanings according to the file system and according to the individual object types supported within a file system. Usage can indicate the opening or closing of a file or can refer to adding links, renaming, restoring, or checking out an object. The usage information format is defined in the Qp0lstdi.h header file as data type Qp0l_Usage_t and is shown in the following table. This attribute can be set to zero only. An attempt to set to any other value will result in *errno* [EINVAL].

When this attribute is set, the date use count reset for the object is set to the current date.

300 QP0L_ATTR_SUID: (CHAR(1)) Set effective user ID (UID) at execution time. This value is ignored if the specified object is a directory. Valid values are:

x'00' QP0L_SUID_OFF: The user ID (UID) is not set at execution time.

x'01' QP0L_SUID_ON: The object owner is the effective user ID (UID) at execution time.

301 QP0L_ATTR_SGID: (CHAR(1)) Set effective group ID (GID) at execution time. Valid values are:

x'00' QP0L_SGID_OFF: If the object is a file, the group ID (GID) is not set at execution time. If the object is a directory in the "root" (/), QOpenSys, and user-defined file systems, the group ID (GID) of objects created in the directory is set to the effective GID of the thread creating the object. This value cannot be set for other file systems.

x'01' QP0L_SGID_ON: If the object is a file, the group ID (GID) is set at execution time. If the object is a directory, the group ID (GID) of objects created in the directory is set to the GID of the parent directory.

Offset to next attribute entry. (Output) This field is not used by the **Qp0lSetAttr()** function. It is provided for alignment so that the same buffer format returned from the **Qp0lGetAttr()** function can be used as input to the **Qp0lSetAttr()** function.

Reserved. A reserved field. This field must be set to binary zero.

Size of attribute data. The exact size of the data for this attribute. If this size does not match the size that the system stores for this attribute, [EINVAL] is returned.

Buffer_Size
(Input) The size in bytes of the buffer pointed to by the *Buffer_ptr* parameter.

Follow_Symlnk
(Input) If the last component in the **Path_Name* is a symbolic link, **Qp0lSetAttr()** either acts upon the symbolic link or the path contained in the symbolic link. This depends on the value of the *Follow_Symlnk* parameter. Valid values are:

0 QP0L_DONOT_FOLLOW_SYMLNK: A symbolic link in the last component is not followed. Attributes of the symbolic link object are set.

1 QP0L_FOLLOW_SYMLNK: A symbolic link in the last component is followed. The attributes of the object contained in the symbolic link are set.

Authorities

Note: Adopted authority is not used.

Authorization Required for Qp0lSetAttr() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)		
Object Referred to	**Authority Required**	**errno**
Each directory, preceding the last component, in the *path name* » except when setting the QP0L_ATTR_CRTOBJAUD attribute. «	*X	EACCES
Object, when setting the QP0L_ATTR_RESET_DATE, QP0L_ATTR_ALWCKPWRT, QP0L_ATTR_ALWSAV, QP0L_ATTR_DISK_STG_OPT or QP0L_ATTR_MAIN_STG_OPT attribute	*OBJMGT	EACCES
Object, when setting the QP0L_ATTR_CREATE_TIME, QP0L_ATTR_ACCESS_TIME, or QP0L_ATTR_MODIFY_TIME attribute to the current time	Owner or *W (See **Note**)	EACCES
Object, when setting the QP0L_ATTR_RSTDRNMUNL, QP0L_ATTR_SUID, or QP0l_ATTR_SGID values	Owner (See **Note**)	EPERM
Object, when setting the QP0L_ATTR_CREATE_TIME, QP0L_ATTR_ACCESS_TIME, or QP0L_ATTR_MODIFY_TIME attribute to a specific time	*W	EPERM
User, when setting the QP0L_ATTR_CRTOBJSCAN or QP0L_ATTR_SCAN attribute	*ALLOBJ, *SECADM	EPERM
» User, when setting the QP0L_ATTR_CRTOBJAUD attribute	*AUDIT	EPERM «
Object, when setting any other attribute	*W	EACCES

Authorization Required for Qp0lSetAttr() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)		
Object Referred to	**Authority Required**	**errno**
Note: If the file system supports *ALLOBJ special authority and if you have *ALLOBJ special authority, you do not need the listed object authority.		

Authorization Required for Qp0lSetAttr() in the QSYS.LIB and independent ASP QSYS.LIB File Systems		
Object Referred to	**Authority Required**	**errno**
Each directory, preceding the last component, in the *path name* except when setting the QP0L_ATTR_CRTOBJAUD attribute.	*X	EACCES
Object, when setting the QP0L_ATTR_RESET_DATE attribute and the object type is *FILE	*OBJOPR and *OBJMGT	EACCES or EPERM
Object, when setting the QP0L_ATTR_RESET_DATE attribute and the object is a database file member	*X and *OBJMGT	EACCES or EPERM
Object, when setting the QP0L_ATTR_RESET_DATE attribute and the object is neither a *FILE object type nor a database file member	*OBJMGT	EACCES or EPERM
User, when setting the QP0L_ATTR_CRTOBJAUD attribute	*AUDIT	EPERM

Authorization Required for Qp0lSetAttr() in the QDLS File System		
Object Referred to	**Authority Required**	**errno**
Each directory, preceding the last component, in the *path name*	*X	EACCES
Object, when setting the QP0L_ATTR_RESET_DATE attribute	*W, *OBJMGT (See **Note**)	EACCES
Object, when setting any other attribute	*W (See **Note**)	EACCES
Note: If you have *ALLOBJ special authority, you do not need the listed object authority.		

Return Value

0 The **Qp0lSetAttr() API** was successful.

-1 The **Qp0lSetAttr() API** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If the **Qp0lSetAttr() API** is not successful, *errno* indicates one of the following errors:

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	

Error condition	Additional information
[EBUSY (page 540)]	
[ECANCEL (page 543)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EOFFLINE (page 545)]	
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESCANFAILURE (page 547)]	You have attempted to modify an object that has been marked as a scan failure due to processing by an exit program associated with the scan-related integrated file system exit points.
[EUNKNOWN (page 544)]	

Additionally, if interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPE3418 E	Possible APAR condition or hardware failure.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. "Root" (/), QOpenSys, and User-Defined File System Differences

   The QP0L_ATTR_CREATE_TIME and QP0L_ATTR_RESET_DATE attributes are supported for objects of type *STMF only. Attempts to set them on other objects will result in the operation failing with *errno* set to [ENOTSUP].

   The QP0L_ALWSAV_YES value cannot be specified for the QP0L_ATTR_ALWSAV attribute for /dev/null, /dev/zero or objects of type *SOCKET. Attempts to set it on these objects will result in the operation failing with *errno* set to [ENOTSUP].

   The QP0L_ATTR_SGID attribute of the directory affects what the group ID (GID) is for objects that are created in the directory. If the QP0L_ATTR_SGID attribute of the parent directory is off, the group ID (GID) is set to the effective GID of the thread creating the object. If the QP0L_ATTR_SGID attribute of the parent directory is on, the group ID (GID) of the new object is set to the GID of the parent directory. For all other file systems, the GID of the new object is set to the GID of the parent directory.

   When setting the QP0L_ATTR_RSTDRNMUNL, QP0L_ATTR_SUID, or QP0L_ATTR_SGID attributes on an object that has a primary group, it must match the primary group ID or one of the supplemental group IDs of the caller of this API; otherwise, the QP0L_ATTR_SGID attribute is set to QP0L_SGID_OFF.
3. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   The following attributes may be set on objects in these file systems:
   - QP0L_ATTR_CRTOBJAUD
   - QP0L_ATTR_RESET_DATE

     When you set the QP0L_ATTR_RESET_DATE attribute of a database file, all members in that file will have their days used count reset to 0 also.

   Attempting to set any other attribute other than QP0L_ATTR_SUID or QP0L_ATTR_SGID will result in the operation failing with *errno* set to [ENOTSUP].

QSYS.LIB and Independent ASP QSYS.LIB do not support setting the QP0L_ATTR_SUID or QP0L_ATTR_SGID attributes. They will be ignored if specified.

4. Network File System Differences

   When you set the following attributes on objects in the Network File System, the operation will fail with the *errno* set to [ENOTSUP] if the attribute is not set to the following attribute value.

   - If set, QP0L_ATTR_PC_READ_ONLY must be set to an attribute value of QP0L_PC_NOT_READ_ONLY.
   - If set, QP0L_ATTR_PC_HIDDEN must be set to an attribute value of QP0L_PC_NOT_HIDDEN.
   - If set, QP0L_ATTR_PC_SYSTEM must be set to an attribute value of QP0L_PC_NOT_SYSTEM.
   - If set, QP0L_ATTR_PC_ARCHIVE must be set to an attribute value of QP0L_PC_NOT_CHANGED; however, if the object is of type *STMF, the attribute value must be QP0L_PC_CHANGED.
   - If set, QP0L_ATTR_SYSTEM_ARCHIVE must be set to an attribute value of QP0L_SYSTEM_NOT_CHANGED.

   The QP0L_ATTR_CREATE_TIME, QP0L_ATTR_RESET_DATE, QP0L_ATTR_CODEPAGE, QP0L_ATTR_CCSID, QP0L_ATTR_ALWSAV, QP0L_ATTR_ALWCKPWRT, QP0L_ATTR_DISK_STG_OPT, QP0L_ATTR_MAIN_STG_OPT » and QP0L_ATTR_CRTOBJAUD « attributes cannot be set on objects within the Network File System or they will result in the operation failing with *errno* set to [ENOTSUP].

   The NFS client supports the QP0L_ATTR_SUID, QP0L_ATTR_SGID, and QP0L_ATTR_RSTDRNMUNL attributes by passing them to the server over the network and surfacing them to the caller. Whether a particular network file system supports the setting of these attributes depends on the server. Most servers have the capability of masking off the QP0L_ATTR_SUID and QP0L_ATTR_SGID attributes if the NOSUID option is specified on the export. The default, however, is to support these attributes.

5. QNetWare File System Differences

   The QNetWare File System does not support setting the QP0L_ATTR_RSTDRNMUNL, QP0L_ATTR_SYSTEM_ARCHIVE, QP0L_ATTR_RESET_DATE, » QP0L_ATTR_CRTOBJAUD attributes. « If you set any attribute on a NetWare Directory Services (NDS) object, the operation will fail with *errno* set to [ENOTSUP].

   QNetWare supports the QP0L_ATTR_SUID and QP0L_ATTR_SGID attributes by passing them to the server and surfacing them to the caller. Some versions of NetWare may support the attributes and others may not.

6. QDLS File System Differences

   The following attributes may be set on objects in this file system:

   - QP0L_ATTR_ACCESS_TIME
   - QP0L_ATTR_CCSID
   - QP0L_ATTR_CODEPAGE
   - QP0L_ATTR_MODIFY_TIME
   - QP0L_ATTR_PC_ARCHIVE
   - QP0L_ATTR_PC_HIDDEN
   - QP0L_ATTR_PC_READ_ONLY
   - QP0L_ATTR_PC_SYSTEM
   - QP0L_ATTR_RESET_DATE (for documents only)

   Attempting to set any other than the QP0L_ATTR_SUID or QP0L_ATTR_SGID attributes will result in the operation failing with *errno* set to [ENOTSUP].

   QDLS does not support setting the QP0L_ATTR_SUID or QP0L_ATTR_SGID attributes. They will be ignored if specified.

7. QOPT File System Differences

If you set the QP0L_ALWSAV_YES value for the QP0L_ATTR_ALWSAV attribute, the operation will fail with *errno* set to [ENOTSUP].

QOPT does not support setting the QP0L_ATTR_SUID, QP0L_ATTR_SGID, QP0L_ATTR_RSTDRNMUNL, or QP0L_ATTR_CRTOBJAUD attributes for any optical media format. If any attribute is specified, the operation will fail with *errno* set to [ENOTSUP].

8. QFileSvr.400 File System Differences

   QFileSvr.400 supports the QP0L_ATTR_SUID, QP0L_ATTR_SGID, and QP0L_ATTR_RSTDRNMUNL attributes by passing them to the server and surfacing them to the caller.

   QFileSvr.400 does not support setting the QP0L_ATTR_ALWSAV attribute. The operation will fail if this attribute is specified.

   QFileSvr.400 supports setting the QP0L_ATTR_CRTOBJAUD attribute. However, the QSECOFR user profiles on the source and target system must be enabled, and their passwords must match for the operation to succeed.

9. QNTC File System Differences

   The following attributes may be set on objects in this file system:

   - QP0L_ATTR_PC_ARCHIVE
   - QP0L_ATTR_PC_HIDDEN
   - QP0L_ATTR_PC_READ_ONLY
   - QP0L_ATTR_PC_SYSTEM

   Attempting to set any other than the QP0L_ATTR_SUID or QP0L_ATTR_SGID attributes will result in the operation failing with *errno* set to [ENOTSUP].

   QNTC does not support setting the QP0L_ATTR_SUID or QP0L_ATTR_SGID attributes. They will be ignored if specified.

Related Information

- The <**Qp0lstdi.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**qlg.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "Integrated File System Scan on Close Exit Program" on page 513
- "Integrated File System Scan on Open Exit Program" on page 523.
- "QlgSetAttr()—Set Attributes (using NLS-enabled path name)" on page 292—Set Attributes (using NLS-enabled path name)
- "Qp0lGetAttr()—Get Attributes" on page 326—Get Attributes
- Retrieve System Values (QWCRSVAL) API

Example

See Code disclaimer information for information pertaining to code examples.

The following is an example showing a call to the **Qp0lSetAttr()** and the **Qp0lGetAttr()** APIs.

```
 /****************************************************************/
#include "Qp0lstdi.h"
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/types.h>


 int GetAttrObject(
                  Qlg_Path_Name_T *Pathname_ptr,
                  char            *Buffer_ptr,
                  unsigned int    Buffer_size)
 {
```

```
/***********************************************************/
/*   Local variables                                       */
/***********************************************************/
 struct attrStruct
 {
   Qp0l_AttrTypes_List_t  attr_struct;
   uint                   AttrTypes[10];
 };
 struct attrStruct  Attr_types_ptr;

 unsigned int buff_size_needed;
 unsigned int num_bytes_returned;
 unsigned int follow_sym;
 int          rc;

/***********************************************************/
/* Start of executable code                                */
/***********************************************************/

/***********************************************************/
/* Initialize Get Attributes Parameters                    */
/***********************************************************/
 memset((void *)&Attr_types_ptr, 0x00,sizeof(struct attrStruct));
 Attr_types_ptr.attr_struct.Number_Of_ReqAttrs = 3;
 Attr_types_ptr.AttrTypes[0] = QP0L_ATTR_PC_READ_ONLY;
 Attr_types_ptr.AttrTypes[1] = QP0L_ATTR_PC_HIDDEN;
 Attr_types_ptr.AttrTypes[2] = QP0L_ATTR_CODEPAGE;
 buff_size_needed = 0;
 follow_sym = QP0L_FOLLOW_SYMLNK;

/***********************************************************/
/* Call Qp0lGetAttr() to retrieve attributes.              */
/***********************************************************/
 rc = Qp0lGetAttr(Pathname_ptr,
                  (Qp0l_AttrTypes_List_t *)&Attr_types_ptr,
                  Buffer_ptr,
                  Buffer_size,
                  &buff_size_needed,
                  &num_bytes_returned,
                  follow_sym);

 if((rc == 0) &&              /* If successful, but          */
    (num_bytes_returned <= 0)) /* Incorrect bytes returned    */
   rc = EUNKNOWN;             /* Unknown error               */

 return(rc);
}                             /* End GetAttrObject()         */


int SetAttrObject(
                 Qlg_Path_Name_T *Pathname_ptr,
                 char            *Buffer_ptr,
                 unsigned int    Buffer_size)
{

/***********************************************************/
/*   Local variables                                       */
/***********************************************************/

 unsigned int       follow_sym;
 int                rc;
 int                done = 0;
 unsigned int       attrSize;
 Qp0l_Attr_Header_t *attrPtr;

/***********************************************************/
/* Start of executable code                                */
```

```
/**************************************************************/

/**************************************************************/
/* Initialize Set Attributes Parameters                       */
/**************************************************************/
 follow_sym = QP0L_FOLLOW_SYMLNK;

/**************************************************************/
/* Qp0lSetAttr() only sets one attribute at a time.  The      */
/* buffer from Qp0lGetAttr may contain more than one          */
/* attribute to set.  We may have to call Qp0lSetAttr()       */
/* multiple times.  The Next_Attr_Offset value is the key.    */
/* If it is greater than zero, then there is another          */
/* attribute in the buffer.  Also, it is important to note    */
/* that the value stored there is the offset from the start   */
/* of the buffer, not the offset from the start of the        */
/* current entry.                                             */
/**************************************************************/
 attrPtr = (Qp0l_Attr_Header_t *)Buffer_ptr;
 while(!done)
 {
     attrSize = attrPtr->Attr_Size +
       sizeof(Qp0l_Attr_Header_t);  /* Calculate attr size    */
     /*******************************************************/
     /* Call Qp0lSetAttr() to set the attribute              */
     /*******************************************************/
     rc=Qp0lSetAttr(Pathname_ptr,
                    (char *)attrPtr,
                    attrSize,
                    follow_sym);
     if(rc != 0)              /* If the function failed       */
       done = 1;              /* End the loop                 */
     else if(attrPtr->Next_Attr_Offset > 0) /* If more data   */
       attrPtr = (Qp0l_Attr_Header_t *) /* Set attribute      */
         (Buffer_ptr + attrPtr->Next_Attr_Offset); /* pointer */
     else                     /* No more data                 */
       done = 1;              /* End the loop                 */
 }
 return(rc);
}                             /* End SetAttrObject()          */


int main (int argc, char *argv[])
{
 #define MYPN "FRED"
 #define MYPN2 "FRED2"
/**************************************************************/
/*   Local variables                                          */
/**************************************************************/
 const char US_const[3] = "US";
 const char Language_const[4] = "ENU";
 const char Path_Name_Del_const[2] = "/";

 typedef struct pnstruct
 {
  Qlg_Path_Name_T  qlg_struct;
  char             pn[sizeof(MYPN)];
 } ;

 typedef struct pnstruct2
 {
  Qlg_Path_Name_T  qlg_struct;
  char             pn[sizeof(MYPN2)];
 } ;

 struct pnstruct pns;
 struct pnstruct2 pns2;
```

```
  int rc;

  char  BufferArea[250];
  unsigned int buffer_size = 250;

 /*************************************************************/
 /* Start of executable code                                  */
 /*************************************************************/

 /*************************************************************/
 /* Initialize Pathname for original object                   */
 /*************************************************************/
  memset((void *)&pns, 0, sizeof(struct pnstruct));
  pns.qlg_struct.CCSID = 37;
  memcpy(pns.qlg_struct.Country_ID,US_const,2);
  memcpy(pns.qlg_struct.Language_ID,Language_const,3);;
  pns.qlg_struct.Path_Type = 0;
  pns.qlg_struct.Path_Length = sizeof(MYPN) - 1;
  memcpy(pns.qlg_struct.Path_Name_Delimiter,Path_Name_Del_const,1);
  memcpy(pns.pn,MYPN,sizeof(MYPN));

 /*************************************************************/
 /* Call GetAttrObject to retrieve attributes from the source */
 /* object.                                                   */
 /*************************************************************/
  rc = GetAttrObject((Qlg_Path_Name_T *)&pns,
                     BufferArea,
                     buffer_size);
  if (rc == 0)                  /* If GetAttr succeeded       */
  {
   /**********************************************************/
   /* Initialize Pathname for target object                  */
   /**********************************************************/
    memset((void *)&pns2, 0, sizeof(struct pnstruct2));
    pns2.qlg_struct.CCSID = 37;
    memcpy(pns2.qlg_struct.Country_ID,US_const,2);
    memcpy(pns2.qlg_struct.Language_ID,Language_const,3);;
    pns2.qlg_struct.Path_Type = 0;
    pns2.qlg_struct.Path_Length = sizeof(MYPN2)-1;
    memcpy(pns2.qlg_struct.Path_Name_Delimiter,Path_Name_Del_const,1);
    memcpy(pns2.pn,MYPN2,sizeof(MYPN2));

   /**********************************************************/
   /* Call SetAttrObject to set attributes on the target     */
   /* object.                                                */
   /**********************************************************/
    rc=SetAttrObject((Qlg_Path_Name_T *)&pns2,
                      BufferArea,
                      buffer_size);
    if (rc != 0)
    {
      rc = errno;              /* return errno from SetAttrObject */
      printf("Qp0lSetAttr() for %s failed with %i.\n",pns2.pn,rc);
    }
  }                            /* end check GetAttrObject rc      */
  else                         /* GetAttrObject failed            */
  {
    rc = errno;                /* return errno from GetAttrObject */
    printf("Qp0lGetAttr() for %s failed with %s.\n",pns.pn,rc);
  }
  return(rc);
}                              /* end main                        */
```

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

Qp0lUnlink()—Remove Link to File

Syntax

```
#include <Qp0lstdi.h>

int Qp0lUnlink(Qlg_Path_Name_T *path_name);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes on "open()—Open File" on page 195.

The **Qp0lUnlink()** function, similar to the **unlink()** function, removes a directory entry that refers to a file. **Qp0lUnlink()** differs from **unlink()** in that the *path_name* parameter is a pointer to a Qlg_Path_Name_T structure instead of a pointer to a character string.

For a discussion of the authorities required, return values, and related information, see "unlink()—Remove Link to File" on page 492.

Parameters

Path_Name
(Input) The path name of the object to be unlinked. This path name is in the Qlg_Path_Name_T format. For more information on this structure, see Path Name Format.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "unlink()—Remove Link to File" on page 492—Remove Link to File
- "link()—Create Link to File" on page 153—Create Link to File
- "open()—Open File" on page 195—Open File
- "close()—Close File or Socket Descriptor" on page 34—Close File or Socket Descriptor
- "rmdir()—Remove Directory" on page 463—Remove Directory

Example

See Code disclaimer information for information pertaining to code examples.

The following example removes a link to a file: This program was stored in a source file with CCSID 37, so the constant string ″newfile″ will be compiled in coded character set identifier (CCSID) 37. Therefore, the country or region and language specified are United States English, and the CCSID specified is 37.

```
#include <fcntl.h>
#include <stdio.h>
#include <Qp0lstdi.h>

main() {
  const char US_const[3]= "US";
  const char Language_const[4]="ENU";
  const char Path_Name_Del_const[2] = "/";

  struct pnstruct
    {
     Qlg_Path_Name_T  qlg_struct;
     char             pn[7];
    };
  struct pnstruct pns;
  struct pnstruct *pns_ptr = NULL;

  char fn[]="unlink.file";
```

```
  memset((void*)&pns, 0x00, sizeof(struct pnstruct));
  pns.qlg_struct.CCSID = 37;
  memcpy(pns.qlg_struct.Country_ID,US_const,2);
  memcpy(pns.qlg_struct.Language_ID,Language_const,3);;
  pns.qlg_struct.Path_Type = 0;
  pns.qlg_struct.Path_Length = sizeof(fn)-1;
  memcpy(pns.qlg_struct.Path_Name_Delimiter,
         Path_Name_Del_const,1);
  memcpy(pns.pn,fn,sizeof(fn));
  memset((void *)&Attr_types_ptr, 0x00,
         sizeof(struct attrStruct));
  pns_ptr = &pns;

  if (Qp0lUnlink((Qlg_Path_Name_T *)&pns) != 0)
      perror("Qp0lunlink() error");
}
```

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

Qp0zPipe()—Create Interprocess Channel with Sockets

Syntax

```
#include <spawn.h>

int Qp0zPipe(int fildes[2]);
```

Service Program Name: QP0ZSPWN
Default Public Authority: *USE
Threadsafe: Yes

The **Qp0zPipe()** function creates a data pipe that can be used by two processes. One end of the pipe is represented by the file descriptor returned in *fildes[0]*. The other end of the pipe is represented by the file descriptor returned in *fildes[1]*. Data that is written to one end of the pipe can be read from the other end of the pipe in a first-in-first-out basis. Both ends of the pipe are open for reading and writing.

The **Qp0zPipe()** function is often used with the **spawn()** function to allow the parent and child processes to send data to each other.

Parameters

fildes[2]
(Input) An integer array of size 2 that will contain the pipe descriptors.

Authorities

None.

Return Value

0	**Qp0zPipe()** was successful.
-1	**Qp0zPipe()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **Qp0zPipe()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EFAULT (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EMFILE (page 543)]	
[ENFILE (page 543)]	
[ENOBUFS (page 542)]	
[EOPNOTSUPP (page 542)]	
[EUNKNOWN (page 544)]	

Usage Notes

The i5/OS implementation of the **Qp0zPipe()**function is based on sockets rather than pipes and, therefore, uses socket descriptors. There are several differences:

1. After calling the **fstat()** function using one of the file descriptors returned on a Qp0zPipe() call, when the `st_mode` from the `stat` structure is passed to the S_ISFIFO() macro, the return value indicates FALSE. When the `st_mode` from the `stat` structure is passed to S_ISSOCK(), the return value indicates TRUE.
2. The file descriptors returned **on a Qp0zPipe()** call can be used with the **send()**, **recv()**, **sendto()**, **recvfrom()**, **sendmsg()**, and **recvmsg()** functions.
3. If this function is called by a thread executing one of the scan-related exit programs (or any of its created threads), the descriptors that are returned are scan descriptors. See "Integrated File System Scan on Open Exit Program" on page 523 and "Integrated File System Scan on Close Exit Program" on page 513 for more information. If a process is spawned, these scan descriptors are not inherited by the spawned process and therefore cannot be used in that spawned process. Therefore, in this case, the descriptors returned by **Qp0zPipe()** function will only work within the same process.

If you want to use the traditional implementation of pipes, in which the descriptors returned are pipe descriptors instead of socket descriptors, use the **pipe()** function.

Related Information

- The <**spawn.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "fstat()—Get File Information by Descriptor" on page 95—Get File Information by Descriptor
- "pipe()—Create an Interprocess Channel" on page 221—Create an Interprocess Channel
- spawn()—Spawn Process
- socketpair()—Create a Pair of Sockets
- "stat()—Get File Information" on page 468—Get File Information

API introduced: V4R1

Top | UNIX-Type APIs | APIs by category

---

qsygetgroups()—Get Supplemental Group IDs

Syntax

```
#include <qsysetid.h>

int qsygetgroups(int gidsetsize, gid_t grouplist[])
```

Service Program Name: QSYSETIDS
Default Public Authority: *USE
Threadsafe: No

If the *gidsetsize* argument is zero, **qsygetgroups()** returns the number of supplemental group IDs associated with the calling thread without modifying the array pointed to by the *grouplist* argument. Otherwise, **qsygetgroups()** fills in the array *grouplist* with the supplementary group IDs of the calling thread and returns the actual number of group IDs stored. The values of array entries with indexes larger than or equal to the returned value are undefined.

Parameters

gidsetsize
(Input) The number of elements in the supplied array *grouplist*.

grouplist
(Output) The supplementary group IDs.

Authorities

No authorization is required.

Return Value

0 or > 0	**qsygetgroups()** was successful. If the *gidsetsize* argument is 0, the number of supplementary group IDs is returned. If *gidsetsize* is greater than 0, the array *grouplist* is filled with the supplementary group IDs of the calling thread and the return value represents the actual number of group IDs stored.
-1	**qsygetgroups()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **qsygetgroups()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EINVAL (page 540)]	The *gidsetsize* argument is not equal to zero and is less than the number of group IDs.

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

---

qsysetegid()—Set Effective Group ID

Syntax

```
#include <qsysetid.h>

int qsysetegid(gid_t gid);
```

Service Program Name: QSYSETIDS
Default Public Authority: *USE
Threadsafe: Yes

If *gid* is equal to either the real, effective, saved group ID, or one of the groups in the supplemental group list, **qsysetegid()** sets the effective group ID to *gid*.

If *gid* is not equal to any of the current groups, but the thread has *USE authority to the user profile associated with the *gid*, **qsysetegid()** sets the effective group ID to *gid*.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

gid (Input) Group ID.

This field must contain one of the following values:

0 There is no effective group ID.

1 to 4294967294
The group ID value for the set operation.

Authorities and Locks

User profile associated with *uid* authority
*USE authority is required to the user profile associated with *gid* if *gid* is not equal to the real, effective, saved group IDs or one of the groups in the supplemental group list.

User profile associated with *uid* lock
*SHRRD

Return Value

0 **qsysetegid()** was successful.

-1 **qsysetegid()** was not successful. *errno* is set to indicate the error.

Error Conditions

If **qsysetegid()** is not successful, *errno* indicates one of the following errors.

Error condition	Additional information
[EAGAIN (page 541)]	User profile associated with the *gid* is locked. Try again.
[EINVAL (page 540)]	The value of the *gid* argument is not valid. Following are possible reasons: • Out of range. • Not associated with a user profile.
[EDAMAGE (page 544)]	The user profile associated with the *gid* or an internal system object is damaged.
[ENOTSUP (page 542)]	Operation not supported. The current effective user profile specifies OWNER(*GRPPRF), but the group profile associated with this *gid* is not equal to the user profile's first group and the user's first group is not in the list of supplemental groups.
[EPERM (page 540)]	Operation not permitted. Following are possible reasons: • The thread does not have *USE authority to the user profile associated with the *gid* and the *gid* to be set is not the same as the real, effective, saved group IDs or any of the supplemental groups. • *gid* cannot be set to 0 if there are supplemental groups.
[EUNKNOWN (page 544)]	An unknown error has occurred. Check the joblog for error messages.

API introduced: V4R5

qsyseteuid()—Set Effective User ID

Syntax

```
#include <qsysetid.h>

int qsyseteuid(uid_t uid);
```

Service Program Name: QSYSETIDS
Default Public Authority: *USE
Threadsafe: Yes

If *uid* is equal to the real, effective, or saved user ID, **qsyseteuid()** sets the effective user ID to *uid*.

If *uid* is not equal to the real, effective, or saved user ID, but the thread has *USE authority to the user profile associated with *uid*, **qsyseteuid()** sets the effective user ID to *uid*.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

uid (Input) User ID.

This field must contain one of the following values:

0 to 4294967294
The user ID value for the set operation.

Authorities and Locks

User profile associated with *uid* authority
*USE authority is required to the user profile associated with *uid* if *uid* is not equal to the real, effective or saved user IDs.

User profile associated with *uid* lock
*SHRRD

Return Value

0 **qsyseteuid()** was successful.

-1 **qsyseteuid()** was not successful. *errno* is set to indicate the error.

Error Conditions

If **qsyseteuid()** is not successful, *errno* indicates one of the following errors.

Error condition	Additional information
[EAGAIN (page 541)]	User profile associated with the *uid* is locked. Try again.
[EDAMAGE (page 544)]	The user profile associated with the *uid* or an internal system object is damaged.
[EINVAL (page 540)]	The value of the *uid* argument is not valid. Following are possible reasons: • Out of range. • Not associated with a user profile.

Error condition	Additional information
[ENOTSUP (page 542)]	Operation not supported. The user profile associated with this *uid* specifies OWNER(*GRPPRF), but the user profile's first group is not the current effective group, nor is it in the list of supplemental groups.
[EPERM (page 540)]	Operation not permitted. The thread does not have *USE authority to the user profile and the *uid* to be set is not the same as the real, effective, or saved user IDs.
[EUNKNOWN (page 544)]	An unknown error has occurred. Check the joblog for error messages.

API introduced: V4R5

Top | UNIX-Type APIs | APIs by category

---

qsysetgid()—Set Group ID

Syntax

```
#include <qsysetid.h>

int qsysetgid(gid_t gid);
```

Service Program Name: QSYSETIDS
Default Public Authority: *USE
Threadsafe: Yes

If the thread has *ALLOBJ special authority, **qsysetgid()** sets the real, effective and saved groups to *gid*.

If the thread does not have *ALLOBJ special authority, but *gid* is equal to the real, effective or saved group IDs, the **qsysetgid()** function sets the effective group ID to *gid*. The real group and saved group IDs remain unchanged.

Any supplementary group IDs of the calling thread remain unchanged.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

gid (Input) Group ID.

This field must contain one of the following values:

0 There is no group ID. The effective group ID can be set to 0 only if there are no supplemental groups.

1 to 4294967294
The group ID value for the set operation.

Authorities and Locks

***ALLOBJ special authority**
*ALLOBJ special authority is required if *gid* is not equal to the real, effective or saved group ID.

User profile associated with *gid* lock
*SHRRD

Return Value

0 **qsysetgid()** was successful.

-1 **qsysetgid()** was not successful. *errno* is set to indicate the error.

Error Conditions

If **qsysetgid()** is not successful, *errno* indicates one of the following errors.

Error condition	Additional information
[EAGAIN (page 541)]	User profile associated with the *gid* is locked. Try again.
[EDAMAGE (page 544)]	The user profile associated with the *gid* or an internal system object is damaged.
[EINVAL (page 540)]	The value of the *gid* argument is not valid. Following are possible reasons: • Out of range. • Not associated with a user profile.
[ENOTSUP (page 542)]	Operation not supported. The current effective user profile specifies OWNER(*GRPPRF), but the group profile associated with this *gid* is not equal to the user profile's first group and the user's first group is not in the list of supplemental groups.
[EPERM (page 540)]	Operation not permitted. Following are possible reasons: • The thread does not have *ALLOBJ special authority and *gid* is not the same as the real, effective or saved group ID. • Tried to set effective group ID to 0 when there are supplemental groups.
[EUNKNOWN (page 544)]	An unknown error has occurred. Check the joblog for error messages.

API introduced: V4R5

Top | UNIX-Type APIs | APIs by category

---

qsysetgroups()—Set Supplemental Group IDs

Syntax

```
#include <qsysetid.h>

int qsysetgroups(int gidsetsize, gid_t grouplist[])
```

Service Program Name: QSYSETIDS
Default Public Authority: *USE
Threadsafe: No

The qsysetgroups API sets the supplementary group IDs of the calling thread. The qsysetgroups API cannot set more than (NGROUPS_MAX-1) groups in the group set.

The real group ID, effective group ID and saved group ID remain unchanged.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

gidsetsize
(Input) The number of elements in the supplied array *grouplist*. If the number of elements is specified as 0, the current supplementary groups will be removed.

grouplist
(Input) The supplementary group IDs.

Authorities and locks

User profile associated with *gid* Authority
*USE authority is required to the user profile associated with each *gid* in the group list if the *gid* is not equal to the current thread's real, effective, or saved group IDs or one of the groups in the current thread's supplemental group list.

User profile associated with each *gid* Lock
*SHRRD

Return Value

0	**qsysetgroups()** was successful.
-1	**qsysetgroups()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **qsysetgroups()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EAGAIN (page 541)]	User profile associated with a *gid* is locked. Try again.
[EDAMAGE (page 544)]	The user profile associated with a *gid* or an internal system object is damaged.
[EINVAL (page 540)]	One of the GID values in the *grouplist* argument is not valid. Following are possible reasons: • Out of range. • Not associated with a user profile. • gidsetsize too large.
[ENOTSUP (page 542)]	Operation not supported. The current effective user profile specifies OWNER(*GRPPRF), but the user's first group is not equal to the current effective group profile and the user's first group is not in this list of supplemental groups.
[EPERM (page 540)]	Operation not permitted. Following are possible reasons: • The thread does not have *USE authority to the user profile associated with the *GID* and the *GID* to be set is not the same as the real, effective, saved group IDs or any of the supplemental groups. • Supplemental groups cannot be set if effective GID is 0.
[EUNKNOWN (page 544)]	An unknown error has occurred. Check the joblog for error messages.

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

qsysetregid()—Set Real and Effective Group IDs

Syntax

```
#include <qsysetids.h>

int qsysetregid(gid_t rgid, gid_t egid);
```

Service Program Name: QSYSETIDS
Default Public Authority: *USE
Threadsafe: Yes

The **qsysetregid()** function is used to set the real and effective group IDs. The real and effective group IDs may be set to different values in the same call.

A thread with *ALLOBJ special authority can set the real group ID and the effective group ID to any valid value.

A thread without *ALLOBJ special authority can only set the real group ID to the saved group ID. A thread without *ALLOBJ special authority can only set the effective group ID to the saved group ID or the real group ID.

Any supplemental group IDs remain unchanged.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

real gid
(Input) Group ID.

This field must contain one of the following values:

0
There is no real group ID.

1 to 4294967294
The group ID value for the set operation.

4294967295
The real group ID does not change. This value is the same as X'FFFFFFFF' or -1 in languages that do not support unsigned integers.

effective gid
(Input) Group ID.

This field must contain one of the following values:

0
There is no effective group ID.

1 to 4294967294
The group ID value for the set operation.

4294967295
The effective group ID does not change. This value is the same as X'FFFFFFFF' or -1 in languages that do not support unsigned integers.

Authorities and Locks

***ALLOBJ special authority**
: *ALLOBJ special authority is required to change the real group ID if *rgid* is not equal to the saved group ID. *ALLOBJ special authority is required to set the effective group ID if the *egid* is not equal to the real group ID or the saved group ID.

User profile associated with *rgid* lock
: *SHRRD

User profile associated with *egid* lock
: *SHRRD

Return Value

0 **qsysetregid()** was successful.

-1 **qsysetregid()** was not successful. The *errno* is set to indicate the error.

Error Conditions

If **qsysetregid()** is not successful, *errno* indicates one of the following errors.

Error condition	Additional information
[EAGAIN (page 541)]	User profile associated with the rgid or rgid is locked. Try again.
[EDAMAGE (page 544)]	The user profile associated with one of the GIDs or an internal system object is damaged.
[EINVAL (page 540)]	The value of the *gid* argument is not valid. Following are possible reasons: • Out of range. • Not associated with a user profile.
[ENOTSUP (page 542)]	Operation not supported. The current effective user profile specifies OWNER(*GRPPRF), but the group profile associated with this *gid* is not equal to the user profile's first group and the user's first group is not in the list of supplemental groups.
[EPERM (page 540)]	Operation not permitted. Following are possible reasons: • The thread does not have *ALLOBJ special authority and a change other than changing the real group ID to the saved group ID, or changing the effective group ID to the real group ID or the saved group ID was requested. • Tried to set effective group ID to 0 when there are supplemental groups.
[EUNKNOWN (page 544)]	An unknown error has occurred. Check the joblog for error messages.

API introduced: V4R5

Top | UNIX-Type APIs | APIs by category

---

qsysetreuid()—Set Real and Effective User IDs

Syntax

```
int qsysetreuid(uid_t ruid, uid_t euid);
```

Service Program Name: QSYSETIDS
Default Public Authority: *USE
Threadsafe: Yes

The **qsysetreuid()** function sets the real and effective user IDs to the values specified by *ruid* and *euid*.

A thread with *ALLOBJ special authority can set either ID to any value.

A thread without *ALLOBJ special authority can only set the effective user ID if the *euid* argument is equal to the real, effective, or saved user ID.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

real uid
(Input) User ID.

This field must contain one of the following values:

0 to 4294967294
The user ID value for the set operation.

4294967295
The real user ID does not change. This value is the same as X'FFFFFFFF' or -1 in languages that do not support unsigned integers.

effective uid
(Input) User ID.

This field must contain one of the following values:

0 to 4294967294
The user ID value for the set operation.

4294967295
The effective user ID does not change. This value is the same as X'FFFFFFFF' or -1 in languages that do not support unsigned integers.

Authorities and Locks

***ALLOBJ special authority**
*ALLOBJ special authority is required to change the real user ID. *ALLOBJ special authorty is required to change the effective user ID if the *euid* is not equal to the real, effective, or saved user ID.

User profile associated with *euid* lock
*SHRRD

User profile associated with *ruid* lock
*SHRRD

Return Value

0 **qsysetreuid()** was successful.

-1 **qsysetreuid()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **qsysetreuid()** is not successful, *errno* indicates one of the following errors.

Error condition	Additional information
[EAGAIN (page 541)]	User profile associated with *ruid* or *euid* is locked. Try again.
[EDAMAGE (page 544)]	The user profile associated with *ruid* or *euid* or an internal system object is damaged.

Error condition	Additional information
[EINVAL (page 540)]	The value of the *ruid* or *euid* argument is not valid. Following are possible reasons: • Out of range. • Not associated with a user profile.
[ENOTSUP (page 542)]	Operation not supported. The user profile associated with this *uid* specifies OWNER(*GRPPRF), but the user profile's first group is not the current effective group, nor is it in the list of supplemental groups.
[EPERM (page 540)]	Operation not permitted. The current thread does not have *ALLOBJ special authority, and either an attempt was made to change the effective user ID to a value other than the real user ID or the saved set-user-ID or an an attempt was made to change the real user ID.
[EUNKNOWN (page 544)]	An unknown error has occurred. Check the joblog for error messages.

API introduced: V4R5

Top | UNIX-Type APIs | APIs by category

qsysetuid()—Set User ID

Syntax

```
#include <qsysetid.h>

int qsysetuid(uid_t uid);
```

Service Program Name: QSYSETIDS
Default Public Authority: *USE
Threadsafe: Yes

If the thread has *ALLOBJ special authority, **qsysetuid()** sets the real, effective, and saved user ID to *uid*.

If the thread does not have *ALLOBJ special authority, but *uid* is equal to the real, effective or saved user ID, **qsysetuid()** sets the effective user ID to *uid*. The real and saved user IDs remain unchanged.

Job scoped locks with a lock state of *SHRRD are held on the user profiles associated with the real user ID, effective user ID, saved user ID, real group ID, effective group ID, saved group ID, and all of the supplemental groups.

Parameters

uid (Input) User ID.

This field must contain one of the following values:

0 to 4294967294
The user ID value for the set operation.

Authorities and Locks

***ALLOBJ special authority**
*ALLOBJ special authority is required if *uid* is not equal to the real, effective, or saved user ID.

User profile associated with *uid* lock
*SHRRD

Return Value

0 **qsysetuid()** was successful.

-1 **qsysetuid()** was not successful. *errno* is set to indicate the error.

Error Conditions

If **qsysetuid()** is not successful, *errno* indicates one of the following errors.

Error condition	Additional information
[EAGAIN (page 541)]	User profile associated with the *uid* is locked. Try again.
[EDAMAGE (page 544)]	The user profile associated with the *uid* or an internal system object is damaged.
[EINVAL (page 540)]	The value of the *gid* argument is not valid. Following are possible reasons: • Out of range. • Not associated with a user profile.
[ENOTSUP (page 542)]	Operation not supported. The user profile associated with this *uid* specifies OWNER(*GRPPRF), but the user profile's first group is not the current effective group, nor is it in the list of supplemental groups.
[EPERM (page 540)]	Operation not permitted. The thread does not have *ALLOBJ special authority and *uid* is not the same as the real, effective or saved user ID.
[EUNKNOWN (page 544)]	An unknown error has occurred. Check the joblog for error messages.

API introduced: V4R5

Top | UNIX-Type APIs | APIs by category

Retrieve Network File System Export Entries (QZNFRTVE) API

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable in bytes	Input	Binary(4)
3	Returned records feedback information	Output	Char(16)
4	Format name	Input	Char(8)
5	Object path name	Input	Char(*)
6	Length of object path name in bytes	Input	Binary(4)
7	CCSID of object path name given	Input	Binary(4)
8	Desired CCSID of the object path names returned	Input	Binary(4)
9	Handle	Input	Binary(4)
10	Error code	I/O	Char(*)

Default Public Authority: *USE
Threadsafe: No

The Retrieve Network File System Export Entries (QZNFRTVE) API returns the list of Network File System (NFS) export entries for objects currently exported to NFS clients or for objects referenced in the /etc/exports file.

Authorities and Locks

- The user must have execute (*X) data authority to the /etc directory (if it exists).
- The user must have read (*R) data authority to the /etc/exports file (if it exists).

Note: Adopted authority is not used.

Usage Notes

If none of the required parameters are passed to this API, then all of the entries that are currently exported will be returned to the joblog by messages (CPIB41A). If there are no entries currently exported, then message CPIB41B will be returned.

Required Parameter Group

The following parameters are required.

Receiver variable
OUTPUT; CHAR(*)

The receiver variable that receives the information requested. The API returns only data that the area can hold.

Length of receiver variable
INPUT; BINARY(4)

The length of the receiver variable provided. The length of the receiver variable parameter may be specified up to the size of the receiver variable area specified by the user program.

No partial entries will be returned. If the length of the receiver variable is less than what is required by the format selected, then an error is returned (CPFB419) and the size required will be indicated in the feedback structure.

Returned records feedback information
OUTPUT; CHAR(16)

Information about the entries that are returned in the receiver variable.

For a detailed description of this format, see “Format of Returned Records Feedback Information” on page 435.

Format name
INPUT; CHAR(8)

The name of the format that is used to retrieve NFS export entries.

You can specify one of the following formats:

EXPE0100
Returns information about export entries that are currently exported. These are sometimes called temporary exports. For a detailed description of this format, see “EXPE0100 and EXPE0200 format” on page 433.

EXPE0200
Returns information about export entries that are in the /etc/exports file. These are sometimes called permanent exports. For a detailed description of this format, see “EXPE0100 and EXPE0200 format” on page 433.

Object path name
INPUT; CHAR(*)

The object path name at which to start listing NFS export entries. Possible values follow:

**FIRST*
NFS export entries are returned starting with the first object path name in the NFS export entry list.

**HANDLE*
NFS export entries are returned starting with the object path name that corresponds to the specified handle.

When the receiver variable is not large enough to hold all of the entries in the NFS export entry list, the API returns a non-zero handle in the returned records feedback information parameter. This handle can be used on a subsequent call to the API to continue retrieving NFS export entries with the next object path name in the NFS export entry list.

There is no implied order to the export entries that are returned. While no sorting or sequencing has been done on the returned entries, a complete list will eventually be returned if the *HANDLE option is used.

Object path name
The NFS export entry for the specified object path name is returned.

Length of object path name
INPUT; BINARY(4)

The length of the object path name in bytes. If one of the special values is given for the object path name, then the length should be given for that special value.

CCSID of object path name given
INPUT; BINARY(4)

The CCSID of the object path name given as an input parameter. Possible values follow:

0 The current Default Job CCSID should be used.

value A valid CCSID number.

Desired CCSID of object the path names returned.
INPUT; BINARY(4)

The Desired CCSID of the object path names returned. The output structure will contain the actual CCSID of the returned object path names. This will match the Desired CCSID given as input, if possible. Possible values follow:

0 The current Default Job CCSID should be used.

value A valid CCSID number.

Handle of starting object path name
INPUT; BINARY(4)

The handle returned from a previous call to the QZNFRTVE API.

This parameter should be 0 if *HANDLE was NOT specified for the object path name parameter.

Error code
I/O; CHAR(*)

The structure in which to return error information. For the format of the structure, see Error Code Parameter.

Receiver Variable Description

The following table describes the order and format of the data returned in the receiver variable. For a detailed description of each field, see "Field Descriptions" on page 435.

EXPE0100 and EXPE0200 format

This structure is used to return NFS export information for a single object path name for both the EXPE0100 and the EXPE0200 formats.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Length of entry
4	4	BINARY(4)	Displacement to object path name
8	8	BINARY(4)	Length of object path name
12	C	BINARY(4)	CCSID of object path name
16	10	BINARY(4)	Read-only flag
20	14	BINARY(4)	NOSUID flag
24	18	BINARY(4)	Displacement to read-write host names
28	1C	BINARY(4)	Number of read-write host names
32	20	BINARY(4)	Displacement to root host names
36	24	BINARY(4)	Number of root host names
40	28	BINARY(4)	Displacement to access host names
44	2C	BINARY(4)	Number of access host names
48	30	BINARY(4)	Displacement to host options
52	34	BINARY(4)	Number of host options
56	38	BINARY(4)	Anonymous user ID
60	3C	CHAR(10)	Anonymous User Profile
*	*	CHAR(*)	Object path name
These fields repeat for each host name in the read-write access list.		BINARY(4)	Length of host name entry
		BINARY(4)	Length of host name
		CHAR(*)	Host name
These fields repeat for each host name in the root access list.		BINARY(4)	Length of host name entry
		BINARY(4)	Length of host name
		CHAR(*)	Host name
These fields repeat for each host name in the access list.		BINARY(4)	Length of host name entry
		BINARY(4)	Length of host name
		CHAR(*)	Host name
These fields repeat for each host name in the host options list.		BINARY(4)	Length of host name options entry
		BINARY(4)	Network data file CCSID
		BINARY(4)	Network path name CCSID
		BINARY(4)	Write mode flag
		BINARY(4)	Length of host name
		CHAR(*)	Host name

Returned Records Feedback Information Description

The following table describes the order and format of the data returned in the returned records feedback information parameter. For a detailed description of each field, see “Field Descriptions” on page 435.

Format of Returned Records Feedback Information

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Bytes returned
4	4	BINARY(4)	Bytes available
8	8	BINARY(4)	Number of NFS export entries
12	C	BINARY(4)	Handle

Field Descriptions

Anonymous User ID. The user ID used as the effective user ID for requests from unknown users. Hex value 0xFFFFFFFF (a value of -1 if this were a signed integer) indicates requests from unknown users are not allowed.

Anonymous User Profile. This is the i5/OS User Profile name that is associated with the Anonymous User ID returned. If the Anonymous User ID has the special value of hex value 0xFFFFFFFF (a value of -1 if this were a signed integer), then the Anonymous User Profile will be set to the special value of *NONE.

Bytes available. The number of bytes of data available to be returned to the user in the receiver variable. If all data is returned, bytes available is the same as the number of bytes returned. If the receiver variable was not large enough to contain all of the data, this value is estimated based on the total number of entries in the NFS export entry list that could be returned.

Bytes returned. The number of bytes of data returned to the user in the receiver variable.

CCSID of object path name. The CCSID of the object path name.

Object path name. The path name of the object for which export information is to be returned.

Displacement to access host names. The offset (in bytes) from the beginning of the NFS export entry to the host names in the access list.

Displacement to host options. The offset (in bytes) from the beginning of the NFS export entry to the host options list.

Displacement to object path name. The offset (in bytes) from the beginning of the NFS export entry to the object path name.

Displacement to read-write host names. The offset (in bytes) from the beginning of the NFS export entry to the host names in the read-write access list.

Displacement to root host names. The offset (in bytes) from the beginning of the NFS export entry to the host names in the root access list.

Handle. The handle to be used on a subsequent call to the API to continue retrieving NFS export entries with the next object path name in the NFS export entry list. 0 indicates all remaining NFS export entries have been returned.

Host name. The host name.

Length of entry. The length (in bytes) of the current NFS export entry. The length can be used to access the next entry.

Length of host name. The length (in bytes) of the host name.

Length of host name entry. The length (in bytes) of this host name entry.

Length of host name options entry. The length (in bytes) of this host name options entry.

Length of object path name. The length (in bytes) of the object path name.

Network data file CCSID. The CCSID used for data of the files sent to and received from the specified host name.

Network path name CCSID. The CCSID used for path name components of the files sent to and received from the specified host name.

NOSUID flag. Whether an attempt by the client to enable bits other than the permission bits will be ignored. Possible values follow:

0 An attempt to set bits other than the permission bits will be carried out.

1 An attempt to set bits other than the permission bits will be ignored.

Number of access host names. The number of host names in the access list.

Number of host options. The number of entries in the host options list.

Number of NFS export entries. The number of complete entries returned in the list of NFS export entries. A value of zero is returned if the list is empty relative to the requested starting position.

Number of read-write host names. The number of host names in the read-write access list.

Number of root host names. The number of host names in the root access list.

Read-only flag. Whether the object is exported allowing only read access. Possible values follow:

0 The object is exported allowing read-write access for all client hosts that are not specifically indicated to have read-only access.

1 The object is exported allowing read-only access for all client hosts that are not specifically indicated to have read-write access.

Write mode flag. Whether write requests are handled synchronously or asynchronously. Synchronously means that data will be written to disk immediately. Asynchronously does not guarantee that data is written to disk immediately, and can be used to improve server performance. Possible values follow:

0 Write requests are performed synchronously.

1 Write requests are performed asynchronously.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.

Message ID	Error Message Text
CPF3C90 E	Literal value cannot be changed.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

API introduced: V4R3

Top | UNIX-Type APIs | APIs by category

read()—Read from Descriptor

Syntax

```
#include <unistd.h>

ssize_t read(int file_descriptor,
           void *buf, size_t nbyte);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 440.

From the file or socket indicated by *file_descriptor*, the **read()** function reads *nbyte* bytes of input into the memory area indicated by *buf*. If *nbyte* is zero, **read()** returns a value of zero without attempting any other action.

If *file_descriptor* refers to a "regular file" (a stream file that can support positioning the file offset) or any other type of file on which the job can do an **lseek()** operation, **read()** begins reading at the file offset associated with *file_descriptor*. A successful **read()** changes the file offset by the number of bytes read.

If **read()** is successful and *nbyte* is greater than zero, the access time for the file is updated.

read() is not supported for directories.

If *file_descriptor* refers to a descriptor obtained using the **open()** function with O_TEXTDATA specified, the data is read from the file assuming it is in textual form. The maximum number of bytes on a single read that can be supported for text data is 2,147,483,408 (2GB - 240) bytes. The data is converted from the code page of the file to the code page of the application, job, or system as follows:

- When reading from a true stream file, any line-formatting characters (such as carriage return, tab, and end-of-file) are just converted from one code page to another.
- When reading from record files that are being used as stream files, end-of-line characters are added to the end of the data in each record.

There are some important considerations when the file is open for text conversion and the CCSIDs involved are not strictly single-byte:

- The **read()** will return the exact number of bytes requested. For some CCSIDs, this may mean that partial characters are returned at the end of the user buffer. In this case, the remainder of the character has been read from the file and internally buffered. The next consecutive read() will begin with the remainder of the partial character. However, if an **lseek()** is performed, the buffered data will be discarded. See "lseek()—Set File Read/Write Offset" on page 157 for more information.
- Because of the above consideration and because of the possible expansion or contraction of converted data, applications using the O_CCSID flag should avoid assumptions about data size and the current file offset. For example, a file might have a physical size of 100 bytes, but after an application has read 100 bytes from the file, the current file offset may be 50. In order to read the whole file, the application might have to read 200 bytes or more, depending on the CCSIDs involved.

If O_TEXTDATA was not specified on the **open()**, the data is read from the file without conversion. The application is responsible for handling the data.

In the QSYS.LIB and independent ASP QSYS.LIB file systems, most end-of-file characters are symbolic; that is, they are stored outside the member. When reading:

- If O_TEXTDATA is specified, both symbolic and nonsymbolic end-of-file characters can be seen.
- If O_TEXTDATA is not specified (binary mode), only nonsymbolic end-of-file characters can be seen.

See the *Usage Notes* for "write()—Write to Descriptor" on page 502.

When *file_descriptor* refers to a socket, the **read()** function reads from the socket identified by the socket descriptor.

When attempting to read from an empty pipe or FIFO:

- If no job has the pipe or FIFO open for writing, **read()** return 0 to indicate end-of-file.
- If some job has the pipe or FIFO open for writing and O_NONBLOCK was specified, **read()** will fail and *errno* will be set to [EAGAIN].
- If some job has the pipe or FIFO open for writing and O_NONBLOCK was not specified, **read()** will block the calling thread until some data is written or until the pipe or FIFO is closed by all jobs that had the pipe or FIFO open for writing.

Parameters

file_descriptor
: (Input) The descriptor to be read.

buf
: (Output) A pointer to a buffer in which the bytes read are placed.

nbyte
: (Input) The number of bytes to be read.

Authorities

No authorization is required.

Return Value

value
: **read()** was successful. The value returned is the number of bytes actually read and placed in *buf*. This number is less than or equal to *nbyte*. It is less than *nbyte* only if **read()** reached the end of the file before reading the requested number of bytes. If **read()** is reading a regular file and encounters a part of the file that has not been written (but before the end of the file), **read()** places bytes containing zeros into *buf* in place of the unwritten bytes.

-1
: **read()** was not successful. The *errno* global variable is set to indicate the error. If the value of *nbyte* is greater than SSIZE_MAX, **read()** sets *errno* to [EINVAL].

Error Conditions

If **read()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems. This may occur if *file_descriptor* refers to a socket and the socket is using a connection-oriented transport service, and a *connect()* was previously completed. The thread, however, does not have the appropriate privileges to the objects that were needed to establish a connection. For example, the *connect()* required the use of an APPC device that the thread was not authorized to.
[EAGAIN (page 541)]	If *file_descriptor* refers to a pipe or FIFO that has its O_NONBLOCK flag set, this error occurs if the **read()** would have blocked the calling thread.
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	This may occur if *file_descriptor* refers to a socket that is using a connectionless transport service, is not a socket of type `SOCK_RAW`, and is not bound to an address. The file resides in a file system that does not support large files, and the starting offset of the file exceeds 2GB minus 2 bytes.
[EIO (page 540)]	
[ENOMEM (page 543)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ENXIO (page 541)]	
[EOVERFLOW (page 546)]	The file is a regular file, *nbyte* is greater than 0, the starting offset is before the end-of-file, and the starting offset is greater than or equal to 2GB minus 2 bytes.
[ERESTART (page 547)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

Error condition	Additional information
[ECONNABORTED (page 542)]	This error code indicates that the transport provider ended the connection abnormally because of one of the following: • The retransmission limit has been reached for data that was being sent on the socket. • A protocol error was detected.
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EINTR (page 541)]	
[ENOTCONN (page 542)]	This error code is returned only on sockets that use a connection-oriented transport service.
[ETIMEDOUT (page 543)]	A non-blocking **connect()** was previously completed that resulted in the connection timing out. No connection is established. This error code is returned only on sockets that use a connection-oriented transport service.
[EUNATCH (page 543)]	

Error condition	Additional information
[EWOULDBLOCK (page 541)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT

- Network File System
- QFileSvr.400

2. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   This function will fail with error code [ENOTSAFE] if the object on which this function is operation is a save file and multiple threads exist in the job.

   This function will fail with error code [EIO] if the file specified is a save file and the file does not contain complete save file data.

   The file access time for a database member is updated using the normal rules that apply to database files. At most, the access time is updated once per day.

   If you previously used the integrated file system interface to manipulate a member that contains an end-of-file character, you should avoid using other interfaces (such as the Source Entry Utility or database reads and writes) to manipulate the member. If you use other interfaces after using the integrated file system interface, the end-of-file information will be lost.

3. QOPT File System Differences

   The file access time is not updated on a **read()** operation.

   When reading from files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being read are ignored.

4. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

   Reading and writing to files with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the **fcntl()** API to get and release these locks.

5. QFileSvr.400 File System Differences

   The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.

6. For sockets that use a connection-oriented transport service (for example, sockets with a type of `SOCK_STREAM`), a return value of zero indicates one of the following:
   - The partner program has issued a **close()** for the socket.
   - The partner program has issued a **shutdown()** to disable writing to the socket.
   - The connection is broken and the error was returned on a previously issued socket function.
   - A **shutdown()** to disable reading was previously done on the socket.

7. The following applies to sockets that use a connectionless transport service (for example, a socket with a type of `SOCK_DGRAM`).
   - If a **connect()** has been issued previously, then data can be received only from the address specified in the previous **connect()**.
   - The address from which data is received is discarded, since the **read()** has no address parameter.
   - The entire message must be read in a single read operation. If the size of the message is too large to fit in the user supplied buffer, the remaining bytes of the message are discarded.
   - A returned value of zero indicates one of the following:
     - The partner program has sent a NULL message (a datagram with no user data).
     - A **shutdown()** to disable reading was previously done on the socket.
     - The buffer length specified was zero.

8. For file systems that do not support large files, **read()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **read()** will return [EOVERFLOW] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.
9. Using this function successfully on the /dev/null or /dev/zero character special file results in a return value of zero. In addition, the access time for the file is updated.

Related Information

- The <**limits.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "dup()—Duplicate Open File Descriptor" on page 55—Duplicate Open File Descriptor
- "dup2()—Duplicate Open File Descriptor to Another Descriptor" on page 58—Duplicate Open File Descriptor to Another Descriptor
- "fclear()—Write (Binary Zeros) to Descriptor" on page 77—Write (Binary Zeros) to Descriptor
- "fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)" on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "ioctl()—Perform I/O Control Request" on page 141—Perform I/O Control Request
- "lseek()—Set File Read/Write Offset" on page 157—Set File Read/Write Offset
- "open()—Open File" on page 195—Open File
- "pread()—Read from Descriptor with Offset" on page 223—Read from Descriptor with Offset
- "pread64()—Read from Descriptor with Offset (large file enabled)" on page 228—Read from Descriptor with Offset (large file enabled)
- "pwrite()—Write to Descriptor with Offset" on page 229—Write to Descriptor with Offset
- "pwrite64()—Write to Descriptor with Offset (large file enabled)" on page 234—Write to Descriptor with Offset (large file enabled)
- "readv()—Read from Descriptor Using Multiple Buffers" on page 455—Read from Descriptor Using Multiple Buffers
- recv()—Receive Data
- recvfrom()—Receive Data
- recvmsg()—Receive Data or Descriptors or Both
- "write()—Write to Descriptor" on page 502—Write to Descriptor
- "writev()—Write to Descriptor Using Multiple Buffers" on page 509—Write to Descriptor Using Multiple Buffers

Example

See Code disclaimer information for information pertaining to code examples.

The following example opens a file and reads input:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

main() {
  int ret, file_descriptor, rc;
  char buf[]="Test text";

  if ((file_descriptor = creat("test.output", S_IWUSR))!= 0)
     perror("creat() error");
  else {
```

```
    if (-1==(rc=write(file_descriptor, buf, sizof(buf)-1)))
       perror("write() error");
    if (close(file_descriptor)!= 0)
       perror("close() error");
  }

  if ((file_descriptor = open("test.output", O_RDONLY)) < 0)
    perror("open() error");
  else {
    ret = read(file_descriptor, buf, sizeof(buf)-1));
    buf[ret] = 0x00;
    printf("block read: \n<%s>\", buf);
    if (close(file_descriptor)!= 0)
       perror("close() error");
  }
  if (unlink("test.output")!= 0)
     perror("unlink() error");
}
```

Output:

```
block read:
<Test text>
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

readdir()—Read Directory Entry

Syntax

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 446.

The **readdir()** function returns a pointer to a `dirent` structure describing the next directory entry in the directory stream associated with *dirp*.

A call to **readdir()** overwrites data produced by a previous call to **readdir()** on the same directory stream. Calls for different directory streams do not overwrite the data of each other.

If the call to **readdir()** actually reads the directory, the access time of the directory is updated.

readdir() performs translation if necessary to convert the directory entry name into the CCSID (coded character set identifier) of the job at the time of the call to **opendir()**.

Parameters

dirp
: (Input) A pointer to a `DIR` that refers to the open directory stream to be read. This pointer is returned by **opendir()** (see "opendir()—Open Directory" on page 212—Open Directory).

  See "QlgReaddir()—Read Directory Entry (using NLS-enabled path name)" on page 280—Read Directory Entry for a description and an example of supplying the *dirp* in any CCSID, using a `dirent_lg` structure.

A `dirent` structure has the following contents:

char	d_reserved1[16]	Reserved.
unsigned int	d_fileno_gen_id	The generation ID associated with the file ID.
ino_t	d_fileno	The file ID of the file. This number uniquely identifies the object within a file system.
unsigned int	d_reclen	The length of the directory entry in bytes.
int	d_reserved3	Reserved.
char	d_reserved4[6]	Reserved.
char	d_reserved5[2]	Reserved.
qlg_nls_t	d_nlsinfo	National language information about d_name. The following fields are defined: *int ccsid* CCSID of the characters in the d_name field. *char country_id[2]* Country or region identifier associated with the d_name field. *char language_id[3]* Language identifier associated with the d_name field. *char nls_reserved[3]* Reserved.
unsigned int	d_namelen	The length of the name in bytes, excluding the null terminator.
char	d_name[640]	A string that gives the name of a file in the directory. This string ends in a terminating null, and has a maximum length of {NAME_MAX} bytes, not including the terminating NULL (see "pathconf()—Get Configurable Path Name Variables" on page 216).

Authorities

No authorization is required. Authorization is verified during **opendir()**.

Note: When reading the contents of the /QSYS.LIB directory, user profile (*USRPRF) objects to which the caller does not have any authority (i.e., *EXCLUDE) will not be returned from **readdir()**.

Return Value

value **readdir()** was successful. The value returned is a pointer to a `dirent` structure describing the next directory entry in the directory stream.

NULL pointer
One of the following has occurred:

- **readdir()** reached the end of the directory stream. The *errno* global variable is not changed.
- **readdir()** was not successful. The *errno* is set.

Error Conditions

If **readdir()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADF (page 543)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. The **readdir_r()** API should be used to read a directory when running in a multithreaded job.
2. Save the data from **readdir()**, if required, before calling **closedir()**, because **closedir()** frees the data.
3. If the *dirp* argument passed to **readdir()** does not refer to an open directory stream, **readdir()** returns the [EBADF] error.
4. **readdir()** buffers multiple directory entries to improve performance. This means the directory is not actually read on each call to **readdir()**. As a result, files that are added to the directory after **opendir()** or **rewinddir()** may not be returned on calls to **readdir()**, and files that are removed may still be returned on calls to **readdir()**.
5. **readdir()** also returns directory entries for dot (.) and dot-dot (..) subdirectories.
6. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   Calls to **readdir()** that update the access time of the directory use the normal rules that apply to libraries and database files. At most, the access time is updated once per day.
7. QDLS File System Differences

   The access time of the directory is updated on **opendir()**. The access time is not affected by **readdir()**.

   When objects in QDLS are accessed, the country or region ID and language ID of the directory entry name are always set to the country or region ID and language ID of the system.

   When a **readdir()** operation specifies the /QDLS directory, the user must have *USE authority to each child object of the /QDLS directory (that is, *USE authority to each object immediately below QDLS in the directory hierarchy). A directory entry is returned only for those objects for which the user has *USE authority. If the **readdir()** operation specifies a directory below QDLS, a directory entry is returned for all objects, even if the user does not have *USE authority for some of the objects.
8. QOPT File System Differences

   The access time of the directory is not updated on a **readdir()** operation.

Related Information

- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**dirent.h**> file see "Header Files for UNIX-Type Functions" on page 537)
- "opendir()—Open Directory" on page 212—Open Directory
- "QlgReaddir()—Read Directory Entry (using NLS-enabled path name)" on page 280—Read Directory Entry
- "rewinddir()—Reset Directory Stream to Beginning" on page 461—Reset Directory Stream to Beginning
- "closedir()—Close Directory" on page 37—Close Directory
- "pathconf()—Get Configurable Path Name Variables" on page 216-Get Configurable Path Name Variables

Example

See Code disclaimer information for information pertaining to code examples.

The following example reads the contents of the "root" (/) directory:

```
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <stdio.h>

main() {
  DIR *dir;
  struct dirent *entry;

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
```

```
  else {
    puts("contents of root:");
    while ((entry = readdir(dir)) != NULL)
      printf("  %s\n", entry->d_name);
    closedir(dir);
  }
}
```

Output:

```
contents of root:
  .
  ..
  QSYS.LIB
  QDLS
  QOpenSys
  QOPT
  home
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

readdir_r()—Read Directory Entry

Syntax

```
#include <sys/types.h>
#include <dirent.h>

int readdir_r(DIR *dirp, struct dirent *entry,
              struct dirent **result);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 450.

The **readdir_r()** function initializes the `dirent` structure that is referenced by *entry* to represent the next directory entry in the directory stream that is associated with *dirp*. The **readdir_r()** function then stores a pointer to the *entry* structure at the location referenced by *result*.

The storage pointed to by *entry* must be large enough for a `dirent` structure.

If the call to **readdir_r()** actually reads the directory, the access time of the directory is updated.

The **readdir_r()** function performs translation, if necessary, to convert the directory entry name into the coded character set identifier (CCSID) of the job at the time of the call to **opendir()**.

Parameters

dirp
: (Input) A pointer to a `DIR` that refers to the open directory stream to be read. This pointer is returned by **opendir()** (see "opendir()—Open Directory" on page 212—Open Directory).

  See "QlgReaddir()—Read Directory Entry (using NLS-enabled path name)" on page 280—Read Directory Entry for a description and an example of supplying the *dirp* in any CCSID.

entry
: (Output) A pointer to a `dirent` structure in which the directory entry is to be placed.

result
: (Output) A pointer to a pointer to a `dirent` structure. Upon successfully reading a directory entry, this `dirent` pointer is set to the same value as *entry*. Upon reaching the end of the directory stream, this pointer will be set to NULL.

A `dirent` structure has the following contents:

char	d_reserved1[16]	Reserved.
unsigned int	d_fileno_gen_id	The generation ID associated with the file ID.
ino_t	d_fileno	The file ID of the file. This number uniquely identifies the object within a file system.
unsigned int	d_reclen	The length of the directory entry in bytes.
int	d_reserved3	Reserved.
char	d_reserved4[6]	Reserved.
char	d_reserved5[2]	Reserved.
qlg_nls_t	d_nlsinfo	National language information about d_name. The following fields are defined: *int ccsid* CCSID of the characters in the d_name field. *char country_id[2]* Country or region identifier that is associated with the d_name field. *char language_id[3]* Language identifier that is associated with the d_name field. *char nls_reserved[3]* Reserved.
unsigned int	d_namelen	The length of the name in bytes, excluding the null terminator.
char	d_name[640]	A string that gives the name of a file in the directory. This string ends in a terminating null, and has a maximum length of {NAME_MAX} bytes, not including the terminating NULL (see "pathconf()—Get Configurable Path Name Variables" on page 216).

Authorities

No authorization is required. Authorization is verified during **opendir()**.

Return Value

0 **readdir_r()** was successful. The *result* parameter points to one of the following:

- A pointer to a `dirent` structure that describes the next directory entry in the directory stream. This will be the same value as the *entry* parameter.
- A NULL pointer. **readdir_r()** reached the end of the directory stream. The *errno* global variable is not changed.

error code
readdir_r() was not successful. This value is set to the same value as the *errno* global variable.

Error Conditions

If **readdir_r()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADF (page 543)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. **readdir_r()** is threadsafe only when directed to a directory in a threadsafe file system.
3. If the *dirp* argument that is passed to **readdir_r()** does not refer to an open directory stream, **readdir_r()** returns the [EBADF] error.
4. **readdir_r()** caches multiple directory entries to improve performance. This means the directory is not actually read on each call to **readdir_r()**. As a result, files that are added to the directory after **opendir()** or **rewinddir()** may not be returned on calls to **readdir_r()**, and files that are removed may still be returned on calls to **readdir_r()**.
5. **readdir_r()** also returns directory entries for dot (.) and dot-dot (..) subdirectories.
6. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   Calls to **readdir_r()** that update the access time of the directory use the normal rules that apply to libraries and database files. At most, the access time is updated once per day.
7. QDLS File System Differences

   The access time of the directory is updated on **opendir()**. The access time is not affected by **readdir_r()**.

   When objects in QDLS are accessed, the country or region ID and language ID of the directory entry name are always set to the country or region ID and language ID of the system.

   When a **readdir_r()** operation specifies the /QDLS directory, the user must have *USE authority to each object in the /QDLS directory (that is, *USE authority to each object immediately below QDLS in the directory hierarchy). A directory entry is returned only for those objects for which the user has *USE authority. If the **readdir_r()** operation specifies a directory below QDLS, a directory entry is returned for all objects, even if the user does not have *USE authority for some of the objects.
8. QOPT File System Differences

   The access time of the directory is not updated on a **readdir_r()** operation.

Related Information

- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537) >
- The <**dirent.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "opendir()—Open Directory" on page 212—Open Directory
- "QlgReaddir()—Read Directory Entry (using NLS-enabled path name)" on page 280—Read Directory Entry
- "readdir_r_ts64()—Read Directory Entry" on page 451—Read Directory Entry
- "rewinddir()—Reset Directory Stream to Beginning" on page 461—Reset Directory Stream to Beginning
- "closedir()—Close Directory" on page 37—Close Directory

- “pathconf()—Get Configurable Path Name Variables” on page 216—Get Configurable Path Name Variables

Example

See Code disclaimer information for information pertaining to code examples.

The following example reads the contents of the ″root″ (/) directory:

```
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <stdio.h>

main() {
  int return_code;
  DIR *dir;
  struct dirent entry;
  struct dirent *result;

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
  else {
    puts("contents of root:");
    for (return_code = readdir_r(dir, &entry, &result);
         result != NULL && return_code == 0;
         return_code = readdir_r(dir, &entry, &result))
      printf("  %s\n", entry.d_name);
    if (return_code != 0)
      perror("readdir_r() error");
    closedir(dir);
  }
}
```

Output:

```
contents of root:
  .
  ..
  QSYS.LIB
  QDLS
  QOpenSys
  QOPT
  home
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

readdir_r_ts64()—Read Directory Entry

Syntax

```
#include <sys/types.h>
#include <dirent.h>

int readdir_r_ts64(DIR * __ptr64 dirp,
                   struct dirent * __ptr64 entry,
                   struct dirent * __ptr64 * __ptr64 result);
```

Service Program Name: QP0LLIBTS
Default Public Authority: *USE
Threadsafe: Conditional; see Usage Notes for “readdir_r()—Read Directory Entry” on page 447.

The **readdir_r_ts64()** function initializes the dirent structure that is referenced by *entry* to represent the next directory entry in the directory stream that is associated with *dirp*. **readdir_r_ts64()** differs from **readdir_r()** in that it accepts 8-byte process local pointers.

For a discussion of the parameters, authorities required, return values, related information, usage notes, and an example for the **readdir_r()** API, see "readdir_r()—Read Directory Entry" on page 447.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

readlink()—Read Value of Symbolic Link

Syntax

```
#include <unistd.h>

int readlink(const char *path, char *buf, size_t bufsiz);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 454.

The **readlink()** function places the contents of the symbolic link *path* in the buffer *buf*. The size of the buffer is set by *bufsiz*. The contents of the returned buffer do not include a terminating NULL. When *bufsiz* is 0, the number of bytes contained in the symbolic link is returned and the buffer is unchanged.

If the buffer is too small to contain the contents of the symbolic link, the contents are truncated to the size of the buffer (*bufsiz*).

A successful **readlink()**, where *bufsiz* is greater than zero, sets the access time of the symbolic link.

Parameters

path (Input) A pointer to the null-terminated path name of the symbolic link.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgReadlink()—Read Value of Symbolic Link (using NLS-enabled path name)" on page 284—Read Value of Symbolic Link for a description and an example of supplying the *path* in any CCSID.

buf (Output) A pointer to the area in which the contents of the link should be stored.

This parameter will be returned in the CCSID currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

bufsiz (Input) The size of *buf* in bytes.

Authorities

Note: Adopted authority is not used.

Authorization required for readlink()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object	None	None

Return Value

value **readlink()** was successful.

When *bufsiz* is greater than zero, the value returned is the number of bytes placed in the buffer. When *bufsiz* is zero, the value returned is the number of bytes contained in the symbolic link. The buffer is not changed.

If the return value is equal to *bufsiz*, the entire contents of the symbolic link may not have been returned. You can determine the size of the contents of the symbolic link by using either **lstat()** or **readlink()** with a zero value for *bufsiz*.

-1 **readlink()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **readlink()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	For example, the named file is not a symbolic link.
[EIO (page 540)]	
[EISDIR (page 544)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EROOBJ (page 545)]	

Error condition	Additional information
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - Network File System
     - QFileSvr.400
2. The following file systems do not support **readlink()**.
   - QSYS.LIB
   - Independent ASP QSYS.LIB

- QDLS
- QOPT
- QNetWare
- QNTC

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "lstat()—Get File or Link Information" on page 162—Get File or Link Information
- "QlgReadlink()—Read Value of Symbolic Link (using NLS-enabled path name)" on page 284—Read Value of Symbolic Link
- "stat()—Get File Information" on page 468—Get File Information
- "symlink()—Make Symbolic Link" on page 485—Make Symbolic Link
- "unlink()—Remove Link to File" on page 492—Remove Link to File

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **readlink()**:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main() {
  char fn[]="readlink.file";
  char sl[]="readlink.symlink";
  char buf[30];
  int  file_descriptor;

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
    if (symlink(fn, sl) != 0)
      perror("symlink() error");
    else {
      if (readlink(sl, buf, sizeof(buf)) < 0)
        perror("readlink() error");
      else printf("readlink() returned '%s' for '%s'\n", buf, sl);

      unlink(sl);
    }
    unlink(fn);
  }
}
```

Output:

```
readlink() returned 'readlink.file' for 'readlink.symlink'
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

readv()—Read from Descriptor Using Multiple Buffers

Syntax

```
#include <sys/types.h>
#include <sys/uio.h>

int readv(int descriptor,
          struct iovec *io_vector[],
          int vector_length)
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 458.

The *readv()* function is used to receive data from a file or socket descriptor. *readv()* provides a way for data to be stored in several different buffers (*scatter/gather I/O*).

See “read()—Read from Descriptor” on page 437 for more information related to reading from a descriptor.

Parameters

descriptor
(Input) The descriptor to be read. The descriptor refers to a file or a socket.

io_vector[]
(I/O) The pointer to an array of type **struct iovec**. **struct iovec** contains a sequence of pointers to buffers in which the data to be read is stored. The structure pointed to by the *io_vector* parameter is defined in **<sys/uio.h>**.

```
struct iovec {
   void      *iov_base;
   size_t   iov_len;
}
```

iov_base and *iov_len* are the only fields in *iovec* used by sockets. *iov_base* contains the pointer to a buffer and *iov_len* contains the buffer length. The rest of the fields are reserved.

vector_length
(Input) The number of entries in *io_vector*.

Authorities

No authorization is required.

Return Value

n **readv()** is successful, where n is the number of bytes read.

-1 **readv()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **readv()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems. This may occur if *file_descriptor* refers to a socket and the socket is using a connection-oriented transport service, and a *connect()* was previously completed. The thread, however, does not have the appropriate privileges to the objects that were needed to establish a connection. For example, the *connect()* required the use of an APPC device that the thread was not authorized to.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	This may occur if *file_descriptor* refers to a socket that is using a connectionless transport service, is not a socket of type `SOCK_RAW`, and is not bound to an address. The file resides in a file system that does not support large files, and the starting offset of the file exceeds 2 GB minus 2 bytes.
[EIO (page 540)]	
[ENOMEM (page 543)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[EOVERFLOW (page 546)]	The file is a regular file, *nbyte* is greater than 0, the starting offset is before the end-of-file and is greater than or equal to 2GB minus 2 bytes.
[ERESTART (page 547)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

Error condition	Additional information
[ECONNABORTED (page 542)]	This error code indicates that the transport provider ended the connection abnormally because of one of the following: • The retransmission limit has been reached for data that was being sent on the socket. • A protocol error was detected.
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EINTR (page 541)]	
[ENOTCONN (page 542)]	
[ETIMEDOUT (page 543)]	A non-blocking **connect()** was previously completed that resulted in the connection timing out. No connection is established. This error code is returned only on sockets that use a connection-oriented transport service.
[EUNATCH (page 543)]	
[EWOULDBLOCK (page 541)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. The *io_vector*[] parameter is an array of **struct iovec** structures. When a *readv()* is issued, the system processes the array elements one at a time, starting with *io_vector*[0]. For each element, **iov_len** bytes

of received data are placed in storage pointed to by **iov_base**. Data is placed in storage until all buffers are full, or until there is no more data to receive. Only the storage pointed to by **iov_base** is updated. No change is made to the **iov_len** fields. To determine the end of the data, the application program must use the following:

- The function return value (the total number of bytes received).
- The lengths of the buffers pointed to by **iov_base**.

3. For sockets that use a connection-oriented transport service (for example, sockets with a type of `SOCK_STREAM`), a returned value of zero indicates one of the following:
   - The partner program has issued a *close()* for the socket.
   - The partner program has issued a *shutdown()* to disable writing to the socket.
   - The connection is broken and the error was returned on a previously issued socket function.
   - A *shutdown()* to disable reading was previously done on the socket.
4. The following applies to sockets that use a connectionless transport service (for example, a socket with a type of `SOCK_DGRAM`):
   - If a *connect()* has been issued previously, then data can be received only from the address specified in the previous *connect()*.
   - The address from which data is received is discarded, because the *readv()* has no address parameter.
   - The entire message must be read in a single read operation. If the size of the message is too large to fit in the user-supplied buffers, the remaining bytes of the message are discarded.
   - A returned value of zero indicates one of the following:
     – The partner program has sent a NULL message (a datagram with no user data).
     – A *shutdown()* to disable reading was previously done on the socket.
     – The buffer length specified by the application was zero.
5. For the file systems that do not support large files, **readv()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **readv()** will return [EOVERFLOW] if the starting offset exceeds 2GB minus 2 bytes and file was not opened for large file access.
6. QFileSvr.400 File System Differences

   The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.
7. QOPT File System Differences

   When reading from files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being read are ignored.
8. Using this function successfully on the /dev/null or /dev/zero character special file results in a return value of 0. In addition, the access time for the file is updated.

Related Information

- The <**limits.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "dup()—Duplicate Open File Descriptor" on page 55—Duplicate Open File Descriptor
- "dup2()—Duplicate Open File Descriptor to Another Descriptor" on page 58—Duplicate Open File Descriptor to Another Descriptor
- "fclear()—Write (Binary Zeros) to Descriptor" on page 77—Write (Binary Zeros) to Descriptor
- "fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)" on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)

- “fcntl()—Perform File Control Command” on page 82—Perform File Control Command
- “ioctl()—Perform I/O Control Request” on page 141—Perform I/O Control Request
- “lseek()—Set File Read/Write Offset” on page 157—Set File Read/Write Offset
- “open()—Open File” on page 195—Open File
- “read()—Read from Descriptor” on page 437—Read from Descriptor
- recv()—Receive Data
- recvfrom()—Receive Data
- recvmsg()—Receive Data or Descriptors or Both
- “write()—Write to Descriptor” on page 502—Write to Descriptor
- “writev()—Write to Descriptor Using Multiple Buffers” on page 509—Write to Descriptor Using Multiple Buffers

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

rename()—Rename File or Directory

Syntax

```
#include <Qp0lstdi.h>

int rename(const char *old, const char *new);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 461.

The **rename()** function can be defined to be either **Qp0lRenameUnlink()** or **Qp0lRenameKeep()**, depending upon the definitions of the _POSIX_SOURCE and _POSIX1_SOURCE macros in the **<Qp0lstdi.h>** header file:

- When _POSIX_SOURCE or _POSIX1_SOURCE is defined, **rename()** is defined to be **Qp0lRenameUnlink()**. Either **rename()** or **Qp0lRenameUnlink()** can be used to rename a file or directory with the semantics of **Qp0lRenameUnlink()**.
- When _POSIX_SOURCE and _POSIX1_SOURCE are **not** defined, **rename()** is defined to be **Qp0lRenameKeep()**. Either **rename()** or **Qp0lRenameKeep()** can be used to rename a file or directory with the semantics of **Qp0lRenameKeep()**.

When the **<Qp0lstdi.h>** header file is **not** included, **rename()** operates only on database files in the QSYS.LIB or independent ASP QSYS.LIB file system, as it did before the introduction of the integrated file system.

For details on the use of **rename()**, see the “Qp0lRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists” on page 379 and “Qp0lRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists” on page 373 functions.

Authorities and Locks

None.

Parameters

old (Input) A pointer to the null-terminated path name of the file to be renamed.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

new (Input) A pointer to the null-terminated path name of the new name of the file.

This parameter is assumed to be represented in the CCSID currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

The new file name is assumed to be represented in the language and country or region currently in effect for the process.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400

Related Information

- The <**stdio.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**Qp0lstdi.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “pathconf()—Get Configurable Path Name Variables” on page 216—Get Configurable Path Name Variables
- “Qp0lRenameKeep()—Rename File or Directory, Keep ″new″ If It Exists” on page 373—Rename File or Directory, Keep ″new″ If It Exists
- “Qp0lRenameUnlink()—Rename File or Directory, Unlink ″new″ If It Exists” on page 379—Rename File or Directory, Unlink ″new″ If It Exists

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

rewinddir()—Reset Directory Stream to Beginning

Syntax

```
#include <sys/types.h>
#include <dirent.h>

void rewinddir(DIR *dirp);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Yes

The **rewinddir()** function "rewinds" the position of an open directory stream to the beginning. *dirp* points to a `DIR` associated with an open directory stream.

The next call to **readdir()** reads the first entry in the directory. If the contents of the directory have changed since the directory was opened and **rewinddir()** is called, subsequent calls to **readdir()** read the changed contents.

Parameters

dirp (Input) A pointer to a `DIR` that refers to the open directory stream to be rewound. This pointer is returned by the `opendir()` function.

Authorities

No authorization is required. Authorization is verified during **opendir()**.

Return Value

None.

Error Conditions

None.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF1F05 E	Directory handle not valid.
CPF3CF2 E	Error(s) occurred during running of &1 API.

Usage Notes

1. If the *dirp* argument passed to **rewinddir()** does not refer to an open directory, unexpected results could occur.
2. Files that are added to the directory after **opendir()** or **rewinddir()** may not be returned on calls to **readdir()**.

Related Information

- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**dirent.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "opendir()—Open Directory" on page 212—Open Directory
- "readdir()—Read Directory Entry" on page 443—Read Directory Entry
- "closedir()—Close Directory" on page 37—Close Directory

Example

See Code disclaimer information for information pertaining to code examples.

The following example produces the contents of a directory by opening it, rewinding it, and closing it:

```
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <stdio.h>

main() {
  DIR *dir;
  struct dirent *entry;

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
  else {
    puts("contents of root:");
    while ((entry = readdir(dir)) != NULL)
      printf("%s ", entry->d_name);
    rewinddir(dir);
    puts("");
    while ((entry = readdir(dir)) != NULL)
      printf("%s ", entry->d_name);
    closedir(dir);
    puts("");
  }
}
```

Output:

```
contents of root:
  . .. QSYS.LIB QDLS QOpenSys QOPT home
  . .. QSYS.LIB QDLS QOpenSys QOPT home newdir
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

rmdir()—Remove Directory

Syntax

```
#include <unistd.h>

int rmdir(const char *path);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 466.

The **rmdir()** function removes a directory, *path*, provided that the directory is empty; that is, the directory contains no entries other than "dot" (.) or "dot-dot" (..). *path* must not end in dot (.) or dot-dot (..).

If no job currently has the directory open, **rmdir()** deletes the directory itself. The space occupied by the directory is freed for new use. If one or more jobs have the directory open, **rmdir()** removes the link and the dot (.) or dot-dot (..). entries. The directory itself is not removed until the last job closes the directory. New files cannot be created under a directory after the last link is removed, even if the directory is still open.

rmdir() does not remove a directory that still contains files or subdirectories. If *path* refers to a directory that is not empty, the [ENOTEMPTY] error is returned. If *path* refers to the current directory of the current job, to the "root" (/) directory, or to a directory that cannot be removed, the [EBUSY] error is returned.

If *path* refers to a symbolic link, **rmdir()** does not affect any file or directory named by the contents of the symbolic link.

If **rmdir()** is successful, the change and modification times for the parent directory are updated.

Parameters

path (Input) A pointer to the null-terminated path name of the directory to be removed.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgRmdir()—Remove Directory (using NLS-enabled path name)" on page 290 for a description and an example of supplying the *path* in any CCSID.

Authorities

Note: Adopted authority is not used.

Authorization Required for rmdir() (excluding QSYS.LIB, independent ASP QSYS.LIB, and QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be removed	*X	EACCES
Parent directory of the directory to be removed	*WX	EACCES
Directory to be removed	*OBJEXIST	EACCES
Parent directory of the directory to be removed has the S_ISVTX mode bit set to binary one (see **Note**).	*ALLOBJ, or owner of the directory to be removed, or owner of the parent directory of the directory to be removed	EPERM
Note: The S_ISVTX mode bit (which is equivalent to the 'Restricted rename and unlink' object attribute) restriction only applies to objects in the "root" (/), QOpenSys, and user-defined file systems.		

Authorization Required for rmdir() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be removed	*X	EACCES
Parent directory of the directory to be removed	*X	EACCES
Directory to be removed, if it is a library	*OBJEXIST, *RX	EACCES
Directory to be removed, if it is a database file	*OBJEXIST, *OBJOPR	EACCES

Authorization Required for rmdir() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the directory to be removed	*X	EACCES
Parent directory of the directory to be removed	*X	EACCES
Directory to be removed	*OBJEXIST, *X	EACCES

Authorization Required for rmdir() in the QOPT File System

Object Referred to	Authority Required	errno
Volume authorization list	*CHANGE	EACCES
Each directory in the path name preceding the directory to be removed if volume media format is Universal Disk Format (UDF)	*X	EACCES
Parent directory of the directory to be removed if volume media format is Universal Disk Format (UDF)	*WX	EACCES
Directory to be removed if volume media format is Universal Disk Format (UDF)	*W	EACCES
Directory and parent directories if volume media format is not Universal Disk Format (UDF)	None	None

Return Value

0 **rmdir()** was successful.

-1 **rmdir()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **rmdir()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	The path cannot be removed because it is the current working directory of the current process, or it is currently being used by the system.
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	For example, the last component of *path* is 'dot' or 'dot-dot'.
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	

Error condition	Additional information
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTEMPTY (page 545)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[EUNKNOWN (page 544)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys

  - User-defined
  - QNTC
  - QSYS.LIB
  - Independent ASP QSYS.LIB
  - QOPT
  - Network File System
  - QFileSvr.400
2. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   If one or more jobs have the library or file open, **rmdir()** returns [EBUSY].

   If **rmdir()** is successful, the change and modification times for the parent library are updated only if the ″directory″ being removed is a database file.
3. QDLS File System Differences

   If one or more jobs have the folder open, or are using the folder as their current directory, **rmdir()** returns [EBUSY].
4. QOPT File System Differences

   The change and modification times of the parent directory are not updated.

   If *path* refers to a directory that any job has open, the [EBUSY] error is returned.
5. QNTC File System Differences

   The change and modification times of the parent directory are not updated.

Related Information

- The <**unistd.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “mkdir()—Make Directory” on page 169—Make Directory
- “QlgRmdir()—Remove Directory (using NLS-enabled path name)” on page 290—Remove Directory (using NLS-enabled path name)
- “unlink()—Remove Link to File” on page 492—Remove Link to File

Example

See Code disclaimer information for information pertaining to code examples.

The following example removes a directory:

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

main() {
  char new_dir[]="new_dir";
  char new_file[]="new_dir/new_file";
  int  file_descriptor;

  if (mkdir(new_dir, S_IRWXU|S_IRGRP|S_IXGRP) != 0)
    perror("mkdir() error");
  else if ((file_descriptor = creat(new_file, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
    unlink(new_file);
  }

  if (rmdir(new_dir) != 0)
```

```
    perror("rmdir() error");
  else
    puts("removed!");
}
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

stat()—Get File Information

Syntax

```
#include <sys/stat.h>

int stat(const char *path, struct stat *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 472.

The **stat()** function gets status information about a specified file and places it in the area of memory pointed to by the *buf* argument.

If the named file is a symbolic link, **stat()** resolves the symbolic link. It also returns information about the resulting file.

Parameters

path (Input) A pointer to the null-terminated path name of the file from which information is required.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgStat()—Get File Information (using NLS-enabled path name)" on page 293 for a description and an example of supplying the *path* in any CCSID.

buf (Output) A pointer to the area to which the information should be written.

The information is returned in the following `stat` structure, as defined in the <**sys/stat.h**> header file:

mode_t	st_mode	A bit string indicating the permissions and privileges of the file. Symbols are defined in the <sys/stat.h> header file to refer to bits in a mode_t value; these symbols are listed in "chmod()—Change File Authorizations" on page 22.
ino_t	st_ino	The file ID for the object. This number uniquely identifies the object within a file system. When st_ino and st_dev are used together, they uniquely identify the object on the system.
nlink_t	st_nlink	The number of links to the file. This field will be 65,535 if the value could not fit in the specified nlink_t field. The complete value will be in the st_nlink32 field.
unsigned short	st_reserved2	Reserved.
uid_t	st_uid	The numeric user ID (uid) of the owner of the file.
gid_t	st_gid	The numeric group ID (gid) for the file.

off_t	st_size	Defined as follows for each file type: *Regular File* The number of data bytes in the file. *Directory* The number of bytes allocated to the directory. *Symbolic Link* The number of bytes in the path name stored in the symbolic link. *Local Socket* Always zero. *Operating System Native Object* This value is dependent on the object type.
time_t	st_atime	The most recent time the file was accessed.
time_t	st_mtime	The most recent time the contents of the file were changed.
time_t	st_ctime	The most recent time the status of the file was changed.
dev_t	st_dev	The file system ID to which the object belongs. This number uniquely identifies the file system to which the object belongs. When st_ino and st_dev are used together, they uniquely identify the object on the system. This field will be 4,294,967,295 if the value could not fit in the specified dev_t field. The complete value will be in the st_dev64 field.
size_t	st_blksize	The block size of the file in bytes. This number is the number of bytes in a block of disk unit storage.
unsigned long	st_allocsize	The number of bytes allocated to the file. The allocated size varies by object type and file system. For example, the allocated size includes the object data size as shown in st_size as well as any logically sized extents to accomodate anticipated future requirements for the object data. It may or may not include additional bytes for attribute information.
qp0l_objtype_t	st_objtype	The object type; for example, *STMF or *DIR. Refer to CL Programming topic for a list of the object types.
unsigned short	st_codepage	The code page derived from the CCSID used for the data in the file or the extended attributes of the directory. If the returned value of this field is zero (0), there is more than one code page associated with the st_ccsid. If the st_ccsid is not a supported CCSID, the st_codepage is set equal to the st_ccsid.
unsigned short	st_ccsid	The CCSID used for the data in the file or the extended attributes of the directory.
dev_t	st_rdev	The device ID of the object if the object is a character special file or block special file. This number uniquely identifies the file device. This field will be 4,294,967,295 if the value could not fit in the specified dev_t field. The complete value will be in the st_rdev64 field.
nlink32_t	st_nlink32	The number of links to the file.
dev64_t	st_rdev64	The device ID of the object in 64 bit format. See st_rdev for more information.
dev64_t	st_dev64	The file system ID to which the object belongs in 64 bit format. See st_dev for more information.

unsigned int	st_vfs	The unique mount ID of the file system on which the object is located. Information about each mounted file system can be obtained by using the QP0L_RETRIEVE_MOUNTED_FILE_SYSTEMS option of "Perform File System Operation (QP0LFLOP) API" on page 315. Unlike st_dev and st_dev64, st_vfs identifies a particular instance of a file system. For any single file system, st_dev and st_dev64 will remain the same across multiple mounts. In contrast, st_vfs is incremented whenever a file system is mounted and is different for each mount of a file system. Therefore, the value of st_vfs may change due to any system processing which unmounts and mounts file systems, such as IPL and Reclaim Storage (RCLSTG).
char	st_reserved1[32]	Reserved.
unsigned int	st_ino_gen_id	The generation ID associated with the file ID.

Values of `time_t` are given in terms of seconds since a fixed point in time called the *Epoch*.

You can examine properties of a `mode_t` value from the `st_mode` field using a collection of macros defined in the **<sys/stat.h>** header file. If *mode* is a `mode_t` value, then:

S_ISBLK(mode)
: Is nonzero for block special files

S_ISCHR(mode)
: Is nonzero for character special files

S_ISDIR(mode)
: Is nonzero for directories

S_ISFIFO(mode)
: Is nonzero for pipes and FIFO special files

S_ISREG(mode)
: Is nonzero for regular files

S_ISLNK(mode)
: Is nonzero for symbolic links

S_ISSOCK(mode)
: Is nonzero for local sockets

S_ISNATIVE(mode)
: Is nonzero for operating system native objects

Authorities

Note: Adopted authority is not used.

Authorization Required for stat()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object, if object type is not *USRPRF	None	None
Object, if object type is *USRPRF	Any authority greater than *EXCLUDE	ENOENT

Return Value

0 **stat()** was successful. The information is returned in *buf*.

-1 **stat()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **stat()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EOVERFLOW (page 546)]	The file size in bytes cannot be represented correctly in the structure pointed to by buf (the file is larger than 2GB minus 1 byte).
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	

Error condition	Additional information
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
   - Where multiple threads exist in the job.
   - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. ″Root″ (/), QOpenSys, and User-Defined File System Differences

   The st_allocsize value can be influenced by the setting of the disk storage option attribute. See “Qp0lSetAttr()—Set Attributes” on page 403 for more information.
3. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   The **stat()** function could return zero for the `st_atime` value (in the `stat` structure) under some conditions.

The S_ISDIR(mode) macro will be nonzero for *LIB objects. It will also be nonzero for *FILE objects, unless the object is a save file.

The S_ISREG(mode) macro will be nonzero for *MBR and *USRSPC objects. It will also be nonzero for *FILE objects when the object is a save file.

The S_ISNATIVE(mode) macro will be nonzero for all other object types.

4. QDLS File System Differences

   If the date corresponding to the `st_atime`, `st_mtime`, or `st_ctime` value precedes 1970, **stat()** returns zero for that value. Also, if the specified *path* is /QDLS, **stat()** returns zero for all three values `st_atime`, `st_mtime`, and `st_ctime`.

   The S_ISDIR(mode) macro will be nonzero for *FLR objects.

   The S_ISREG(mode) macro will be nonzero for *DOC objects.

   The S_ISNATIVE(mode) macro will always be zero.

5. QOPT File System Differences

   The value for st_atime will always be zero. The value for st_ctime will always be the creation date and time of the file or directory.

   The user, group, and other mode bits are always on for an object that exists on a volume not formatted in Universal Disk Format (UDF).

   If the object exists on a volume formatted in Universal Disk Format (UDF), the authorization that is checked for the object and preceding directories in the path name follows the rules described in Authorization Required for stat() (page 470), ″.″ If the object exists on a volume formatted in some other media format, no authorization checks are made on the object or on each directory in the path name. The volume authorization list is checked for *USE authority regardless of the media format of the volume.

   stat() on /QOPT will always return 2,147,483,647 for size fields.

   stat() on optical volumes will return the volume capacity or 2,147,483,647, whichever is smaller.

   The file access time is not changed.

6. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

7. QNetWare File System Differences

   The QNetWare file system does not fully support mode bits. See NetWare on iSeries for more information.

8. QFileSvr.400 File System Differences

   The value of st_vfs will always be 0 for remote objects accessed via QFileSvr.400.

9. This function will fail with the [EOVERFLOW] error if the file size in bytes cannot be represented correctly in the structure pointed to by buf (the file is larger than 2GB minus 1 byte).

10. When you develop in C-based languages and this function is compiled with _LARGE_FILES defined, it will be mapped to “stat64()—Get File Information (Large File Enabled)” on page 475. Note that the type of the buf parameter, `struct stat *`, also will be mapped to type `struct stat64 *`.

Related Information

- The <**sys/stat.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**sys/types.h**> file (see “Header Files for UNIX-Type Functions” on page 537)

- “chmod()—Change File Authorizations” on page 22—Change File Authorizations
- “chown()—Change Owner and Group of File” on page 29—Change Owner and Group of File
- “creat()—Create or Rewrite File” on page 40—Create or Rewrite File
- “dup()—Duplicate Open File Descriptor” on page 55—Duplicate Open File Descriptor
- “fclear()—Write (Binary Zeros) to Descriptor” on page 77—Write (Binary Zeros) to Descriptor
- “fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)” on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)
- “fcntl()—Perform File Control Command” on page 82—Perform File Control Command
- “fstat()—Get File Information by Descriptor” on page 95—Get File Information by Descriptor
- “link()—Create Link to File” on page 153—Create Link to File
- “lstat()—Get File or Link Information” on page 162—Get File or Link Information
- “mkdir()—Make Directory” on page 169—Make Directory
- “open()—Open File” on page 195—Open File
- “QlgStat()—Get File Information (using NLS-enabled path name)” on page 293—Get File Information (using NLS-enabled path name)
- » “Perform File System Operation (QP0LFLOP) API” on page 315—Perform file system operation«
- “read()—Read from Descriptor” on page 437—Read from Descriptor
- “readlink()—Read Value of Symbolic Link” on page 452—Read Value of Symbolic Link
- “stat64()—Get File Information (Large File Enabled)” on page 475—Get File Information (Large File Enabled)
- “symlink()—Make Symbolic Link” on page 485—Make Symbolic Link
- “unlink()—Remove Link to File” on page 492—Remove Link to File
- “utime()—Set File Access and Modification Times” on page 497—Set File Access and Modification Times
- “write()—Write to Descriptor” on page 502—Write to Descriptor

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets status information about a file:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>

main() {
  struct stat info;

  if (stat("/", &info) != 0)
    perror("stat() error");
  else {
    puts("stat() returned the following information about root f/s:");
    printf("  inode:   %d\n",   (int) info.st_ino);
    printf(" dev id:   %d\n",   (int) info.st_dev);
    printf("   mode:   %08x\n",       info.st_mode);
    printf("  links:   %d\n",         info.st_nlink);
    printf("    uid:   %d\n",   (int) info.st_uid);
    printf("    gid:   %d\n",   (int) info.st_gid);
  }
}
```

Output: note that the following information will vary from system to system.

```
stat() returned the following information about root f/s:
  inode:   0
 dev id:   1
   mode:   010001ed
  links:   3
    uid:   137
    gid:   500
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

stat64()—Get File Information (Large File Enabled)

Syntax

```
#include <sys/stat.h>

int stat64(const char *path, struct stat64 *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 477.

The **stat64()** function gets status information about a specified file and places it in the area of memory pointed to by the *buf* argument.

If the named file is a symbolic link, **stat64()** resolves the symbolic link. It also returns information about the resulting file.

stat64() is enabled for large files. It is capable of operating on files larger than 2GB minus 1 byte and returning correct sizes.

For additional information about authorities required, error conditions, and examples, see "stat()—Get File Information" on page 468—Get File Information.

Parameters

path (Input) A pointer to the null-terminated path name of the file from which information is required.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgStat64()—Get File Information (large file enabled and using NLS-enabled path name)" on page 295 for a description and an example of supplying the *path* in any CCSID.

buf (Output) A pointer to the area to which the information should be written.

The information is returned in the following stat64 structure, as defined in the <**sys/stat.h**> header file:

mode_t	st_mode	A bit string indicating the permissions and privileges of the file. Symbols are defined in the <sys/stat.h> header file to refer to bits in a mode_t value; these symbols are listed in "chmod()—Change File Authorizations" on page 22.
ino_t	st_ino	The file ID for the object. This number uniquely identifies the object within a file system. When st_ino and st_dev are used together, they uniquely identify the object on the system.
uid_t	st_uid	The numeric user ID (uid) of the owner of the file.

gid_t	st_gid	The numeric group ID (GID) for the file.
off64_t	st_size	Defined as follows for each file type: *Regular File* The number of data bytes in the file. *Directory* The number of bytes allocated to the directory. *Symbolic Link* The number of bytes in the path name stored in the symbolic link. *Local Socket* Always zero. *Operating System Native Object* This value is dependent on the object type.
time_t	st_atime	The most recent time the file was accessed.
time_t	st_mtime	The most recent time the contents of the file were changed.
time_t	st_ctime	The most recent time the status of the file was changed.
dev_t	st_dev	The file system ID to which the object belongs. This number uniquely identifies the file system to which the object belongs. When st_ino and st_dev are used together, they uniquely identify the object on the system. This field will be 4,294,967,295 if the value could not fit in the specified dev_t field. The complete value will be in the st_dev64 field.
size_t	st_blksize	The block size of the file in bytes. This number is the number of bytes in a block of disk unit storage.
nlink_t	st_nlink	The number of links to the file. This field will be 65,535 if the value could not fit in the specified nlink_t field. The complete value will be in the st_nlink32 field.
unsigned short	st_codepage	The code page derived from the CCSID used for the data in the file or the extended attributes of the directory. If the returned value of this field is 0, a code page could not be derived.
unsigned long long	st_allocsize	The number of bytes allocated to the file. The allocated size varies by object type and file system. For example, the allocated size includes the object data size as shown in st_size as well as any logically sized extents to accomodate anticipated future requirements for the object data. It may or may not include additional bytes for attribute information.
unsigned int	st_ino_gen_id	The generation ID associated with the file ID.
qp0l_objtype_t	st_objtype	The object type; for example, *STMF or *DIR. Refer to CL Programming topic for a list of the object types.
char	st_reserved2[5]	Reserved.
dev-t	st_rdev	The device ID of the object if the object is a character special file or block special file. This number uniquely identifies the file device. This field will be 4,294,967,295 if the value could not fit in the specified dev_t field. The complete value will be in the st_rdev64 field.
dev64_t	st_rdev64	The device ID of the object in 64 bit format. See st_rdev for more information.
dev64_t	st_dev64	The file system ID to which the object belongs in 64 bit format. See st_dev for more information.
nlink32_t	st_nlink32	The number of links to the file.

unsigned int	st_vfs	The unique mount ID of the file system on which the object is located. Information about each mounted file system can be obtained by using the QP0L_RETRIEVE_MOUNTED_FILE_SYSTEMS option of "Perform File System Operation (QP0LFLOP) API" on page 315. Unlike st_dev and st_dev64, st_vfs identifies a particular instance of a file system. For any single file system, st_dev and st_dev64 will remain the same across multiple mounts. In contrast, st_vfs is incremented whenever a file system is mounted and is different for each mount of a file system. Therefore, the value of st_vfs may change due to any system processing which unmounts and mounts file systems, such as IPL and Reclaim Storage (RCLSTG).
char	st_reserved1[22]	Reserved.
unsigned short	st_ccsid	The CCSID used for the data in the file or the extended attributes of the directory.

Values of `time_t` are given in terms of seconds since a fixed point in time called the *Epoch*.

You can examine properties of a `mode_t` value from the `st_mode` field using a collection of macros defined in the **<sys/stat.h>** header file. If *mode* is a `mode_t` value, then:

S_ISBLK(mode)
: Is nonzero for block special files

S_ISCHR(mode)
: Is nonzero for character special files

S_ISDIR(mode)
: Is nonzero for directories

S_ISFIFO(mode)
: Is nonzero for pipes and FIFO special files

S_ISREG(mode)
: Is nonzero for regular files

S_ISLNK(mode)
: Is nonzero for symbolic links

S_ISSOCK(mode)
: Is nonzero for local sockets

S_ISNATIVE(mode)
: Is nonzero for operating system native objects

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use either the **stat64()** API or the **QlgStat64()** API and the `struct stat64` data type, you must compile the source with `_LARGE_FILE_API` defined.
2. All of the usage notes for **stat()** also apply to **stat64()** and to **QlgStat64()**. See "Usage Notes" on page 472 in the **stat()** API.

API introduced: V4R4

statvfs()—Get File System Information

Syntax

```
#include <sys/statvfs.h>

int statvfs(const char *path, struct statvfs *buf);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 481.

The **statvfs()** function gets status information about the file system that contains the file named by the *path* argument. The information will be placed in the area of memory pointed to by the *buf* argument.

If the named file is a symbolic link, **statvfs()** resolves the symbolic link.

Parameters

path (Input) A pointer to the null-terminated path name of the file from which file system information is required.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgStatvfs()—Get File System Information (using NLS-enabled path name)" on page 296 for a description and an example of supplying the *path* in any CCSID.

buf (Output) A pointer to the area to which the information should be written.

The information is returned in the following `statvfs` structure, as defined in the <**sys/statvfs.h**> header file. Signed fields of the `statvfs` structure that are not supported by the mounted file system will be set to -1.

unsigned long	f_bsize	The file system block size in bytes. This number is the number of bytes in a block of disk unit storage. Some file systems may return zero in this field. If this field is zero, then the contents of the `f_blocks`, `f_bfree`, and `f_bavail` fields are undefined.
unsigned long	f_frsize	The fundamental file system block size in bytes. Some file systems may return zero in this field. If this field is zero, then the contents of the `f_blocks`, `f_bfree`, and `f_bavail` fields are undefined.
_Bin8	f_blocks	The total number of blocks in the file system in terms of `f_frsize`.
_Bin8	f_bfree	The total number of free blocks in the file system.
_Bin8	f_bavail	The total number of free blocks available to a non-privileged process.
unsigned long	f_files	The total number of file serial numbers.
unsigned long	f_ffree	The total number of free file serial numbers.
unsigned long	f_favail	The number of free file serial numbers available to a non-privileged process.
unsigned long	f_fsid	The file system ID. This field will be 4,294,967,295 if the value could not fit in the specified unsigned long field.
unsigned long	f_flag	File system flags. Symbols are defined in the <**sys/statvfs.h**> header file to refer to bits in this field (see "The f_flags field" on page 479).

unsigned long	f_namemax	The maximum file name length in the file system. Some file systems may return the maximum value that can be stored in an `unsigned long` to indicate the file system has no maximum file name length. The maximum value that can be stored in an `unsigned long` is defined in <**limits.h**> as ULONG_MAX. This value is the number of bytes allowed in the file name if it were encoded in the CCSID of the job. If the CCSID is mixed, this number is an estimate and may be larger than the actual allowable maximum.
unsigned long	f_pathmax	The maximum path length in the file system. Some file systems may return the maximum value that can be stored in an `unsigned long` to indicate the file system has no maximum path length. The maximum value that can be stored in an `unsigned long` is defined in <**limits.h**> as ULONG_MAX. This value is the number of bytes allowed in the file name if it were encoded in the CCSID of the job. If the CCSID is mixed, this number is an estimate and may be larger than the actual allowable maximum.
long	f_objlinkmax	The maximum number of hard links for objects other than directories.
long	f_dirlinkmax	The maximum number of hard links for a directory.
char	f_reserved1[4]	Reserved.
unsigned long long	f_fsid64	The file system ID in 64 bit format.
char	f_basetype[80]	The NULL-terminated file system type name. The text in this field will be returned in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this is assumed to be represented in the default CCSID of the job.

Warning: Temporary Level 4 Header

The f_flags field: The following symbols are defined in the <**sys/statvfs.h**> header file to refer to bits that may be returned in the `f_flags` field:

ST_RDONLY
The file system is mounted for read-only access.

ST_NOSUID
The file system does not support setuid/setgid semantics.

ST_CASE_SENSITIVE
The file system is case sensitive.

ST_CHOWN_RESTRICTED
The file system restricts the changing of the owner or primary group to a process that has the appropriate privileges.

ST_THREAD_SAFE
The file system is thread-safe. Thread-safe APIs may operate on objects in this file system in a thread-safe manner.

ST_DYNAMIC_MOUNT
The file system allows itself to be dynamically mounted and unmounted.

ST_NO_MOUNT_OVER
The file system does not allow any part of it to be mounted over.

ST_NO_EXPORTS
The file system does not allow any of its objects to be exported to the Network File System (NFS) Server.

ST_SYNCHRONOUS
The file system supports the ″synchronous write″ semantic of NFS Version 2.

Authorities

Note: Adopted authority is not used.

Authorization Required for statvfs()

Object Referred to	Authority Required	errno
Each directory in the path name that precedes the object	*X	EACCES
Object	None	None

Return Value

0 **statvfs()** was successful. The information is returned in *buf*.

-1 **statvfs()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **statvfs()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[EPERM (page 540)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. ″Root″ (/) and QOpenSys File System Differences

   These file systems return the `f_flag` field with the `ST_NOSUID` flag bit turned off. However, support for the setuid/setgid semantics is limited to the ability to store and retrieve the `S_ISUID` and `S_ISGID` flags when these file systems are accessed from the Network File System server.

3. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)
4. When you develop in C-based languages and this function is compiled with _LARGE_FILES defined, it will be mapped to **statvfs64()**. Additionally, the `struct statvfs` data type will be mapped to a `struct statvfs64`.

Related Information

- The <**sys/statvfs.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**sys/types.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "chown()—Change Owner and Group of File" on page 29—Change Owner and Group of File
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "dup()—Duplicate Open File Descriptor" on page 55—Duplicate Open File Descriptor
- "fclear()—Write (Binary Zeros) to Descriptor" on page 77—Write (Binary Zeros) to Descriptor
- "fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)" on page 82—Write (Binary Zeros) to Descriptor
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "fstatvfs()—Get File System Information by Descriptor" on page 101—Get File System Information by Descriptor
- "link()—Create Link to File" on page 153—Create Link to File
- "open()—Open File" on page 195—Open File
- "QlgStatvfs()—Get File System Information (using NLS-enabled path name)" on page 296—Get File System Information (using NLS-enabled path name)
- "read()—Read from Descriptor" on page 437—Read from Descriptor
- "statvfs64()—Get File System Information (64-Bit Enabled)" on page 483—Get File System Information (64-Bit Enabled)
- "unlink()—Remove Link to File" on page 492—Remove Link to File
- "utime()—Set File Access and Modification Times" on page 497—Set File Access and Modification Times
- "write()—Write to Descriptor" on page 502—Write to Descriptor

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets status information about a file system:

```
#include <sys/statvfs.h>
#include <stdio.h>

main() {
  struct statvfs info;

  if (-1 == statvfs("/", &info))
    perror("statvfs() error");
  else {
    puts("statvfs() returned the following information");
    puts("about the root (/) file system:");
    printf("  f_bsize    : %u\n", info.f_bsize);
    printf("  f_blocks   : %08X%08X\n",
```

```
                        *((int *)&info.f_blocks[0]),
                        *((int *)&info.f_blocks[4]));
    printf("  f_bfree    : %08X%08X\n",
                        *((int *)&info.f_bfree[0]),
                        *((int *)&info.f_bfree[4]));
    printf("  f_files    : %u\n", info.f_files);
    printf("  f_ffree    : %u\n", info.f_ffree);
    printf("  f_fsid     : %u\n", info.f_fsid);
    printf("  f_flag     : %X\n", info.f_flag);
    printf("  f_namemax  : %u\n", info.f_namemax);
    printf("  f_pathmax  : %u\n", info.f_pathmax);
    printf("  f_basetype : %s\n", info.f_basetype);
  }
}
```

Output: The following information will vary from file system to file system.

```
statvfs() returned the following information
about the root (/) file system:
  f_bsize    : 4096
  f_blocks   : 00000000002BF800
  f_bfree    : 0000000000091703
  f_files    : 4294967295
  f_ffree    : 4294967295
  f_fsid     : 0
  f_flag     : 1A
  f_namemax  : 255
  f_pathmax  : 4294967295
  f_basetype : "root" (/)
```

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

statvfs64()—Get File System Information (64-Bit Enabled)

Syntax

```
#include <sys/statvfs.h>

int statvfs64(const char *path, struct statvfs64 *buf)
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 485.

The **statvfs64()** function gets status information about the file system that contains the file named by the *path* argument. The information is placed in the area of memory pointed to by the *buf* argument.

If the named file is a symbolic link, **statvfs64()** resolves the symbolic link.

For details about authorities required, error conditions, and examples, see "statvfs()—Get File System Information" on page 478—Get File System Information.

Parameters

path (Input) A pointer to the null-terminated path name of the file from which file system information is required.

This parameter is assumed to be represented in the coded character set identifier (CCSID) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See “QlgStatvfs64()—Get File System Information (64-Bit enabled and using NLS-enabled path name)” on page 298 for a description and an example of supplying the *path* in any CCSID.

buf (Output) A pointer to the area to which the information should be written.

The information is returned in the following `statvfs64` structure, as defined in the <**sys/statvfs.h**> header file. Signed fields of the `statvfs64` structure that are not supported by the mounted file system will be set to -1.

unsigned long	f_bsize	The file system block size in bytes. This number is the number of bytes in a block of disk unit storage. Some file systems may return zero in this field. If this field is zero, then the contents of the `f_blocks`, `f_bfree`, and `f_bavail` fields are undefined.
unsigned long	f_frsize	The fundamental file system block size in bytes. Some file systems may return zero in this field. If this field is zero, then the contents of the `f_blocks`, `f_bfree`, and `f_bavail` fields are undefined.
unsigned long long	f_blocks	The total number of blocks in the file system in terms of `f_frsize`.
unsigned long long	f_bfree	The total number of free blocks in the file system.
unsigned long long	f_bavail	The total number of free blocks available to a non-privileged process.
unsigned long	f_files	The total number of file serial numbers.
unsigned long	f_ffree	The total number of free file serial numbers.
unsigned long	f_favail	The number of free file serial numbers available to a non-privileged process.
unsigned long	f_fsid	The file system ID. This field will be 4,294,967,295 if the value could not fit in the specified unsigned long field.
unsigned long	f_flag	File system flags. Symbols are defined in the <**sys/statvfs.h**> header file to refer to bits in this field (see “The f_flags field” on page 479).
unsigned long	f_namemax	The maximum file name length in the file system. Some file systems may return the maximum value that can be stored in an `unsigned long` to indicate the file system has no maximum file name length. The maximum value that can be stored in an `unsigned long` is defined in <**limits.h**> as ULONG_MAX. This value is the number of bytes allowed in the file name if it were encoded in the CCSID of the job. If the CCSID is mixed, this number is an estimate and may be larger than the actual allowable maximum.
unsigned long	f_pathmax	The maximum path length in the file system. Some file systems may return the maximum value that can be stored in an `unsigned long` to indicate the file system has no maximum path length. The maximum value that can be stored in an `unsigned long` is defined in <**limits.h**> as ULONG_MAX. This value is the number of bytes allowed in the file name if it were encoded in the CCSID of the job. If the CCSID is mixed, this number is an estimate and may be larger than the actual allowable maximum.
long	f_objlinkmax	The maximum number of hard links for objects other than directories.
long	f_dirlinkmax	The maximum number of hard links for a directory.
char	f_reserved1[4]	Reserved.
unsigned long long	f_fsid64	The file system ID in 64 bit format.
char	f_basetype[80]	The NULL-terminated file system type name. The text in this field will be returned in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this is assumed to be represented in the default CCSID of the job.

For further details about the f_flags field, see “statvfs()—Get File System Information” on page 478.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the **statvfs64()** API or the **QlgStatvfs64()** API and the `struct statvfs64` data type, you must compile the source with the _LARGE_FILE_API macro defined.
2. All of the usage notes for **statvfs()** apply to **statvfs64()** and **QlgStatvfs64()**. See “Usage Notes” on page 481 in the **statvfs()** API.

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

---

symlink()—Make Symbolic Link

Syntax

```
#include <unistd.h>

int symlink(const char *pname, const char *slink);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 487.

The **symlink()** function creates the symbolic link named by *slink* with the value specified by *pname*. File access checking is not performed on the file *pname*, and the file need not exist. In addition, a symbolic link can cross file system boundaries.

If *slink* names a symbolic link, **symlink()** fails with the [EEXIST] error.

A symbolic link path name is resolved in the following manner:

- When a component of a path name refers to a symbolic link rather than to a directory, the path name contained in the symbolic link is resolved.
- If the path name in the symbolic link begins with / (slash), the symbolic link path name is resolved relative to the root directory for the job.

  If the path name in the symbolic link does not start with / (slash), the symbolic link path name is resolved relative to the directory that contains the symbolic link.
- If the symbolic link is the last component of a path name, it may or may not be resolved. Resolution depends on the function using the path name. For example, **rename()** does not resolve a symbolic link when the symbolic link is the final component of either the new or old path name. However, **open()** does resolve a symbolic link when the link is the last component.
- If the symbolic link is not the last component of the original path name, remaining components of the original path name are resolved relative to the symbolic link.
- When a / (slash) is the last component of a path name and it is preceded by a symbolic link, the symbolic link is always resolved.

Any files and directories to which a symbolic link refers are checked for access permission.

symlink() sets the access, change, modification, and creation times for the new symbolic link. It also sets the change and modification times for the directory that contains the new symbolic link.

Parameters

pname (Input) A pointer to the null-terminated value of the symbolic link.

The value of the symbolic link is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgSymlink()—Make Symbolic Link (using NLS-enabled path name)" on page 300 for a description and an example of supplying the *pname* in any CCSID.

slink (Input) A pointer to the null-terminated name of the symbolic link to be created.

This parameter is assumed to be represented in the CCSID, language, and country or region currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgSymlink()—Make Symbolic Link (using NLS-enabled path name)" on page 300 for a description and an example of supplying the *slink* in any CCSID.

Authorities

Note: Adopted authority is not used.

Authorization Required for symlink()

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be created	*X	EACCES
Parent directory of object to be created	*WX	EACCES

Return Value

0 **symlink()** was successful.

-1 **symlink()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **symlink()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EEXIST (page 543)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	

Error condition	Additional information
[EINVAL (page 540)]	
[EIO (page 540)]	
[EISDIR (page 544)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOSYS (page 544)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. The following file systems do not support **symlink()**:
   - QSYS.LIB
   - Independent ASP QSYS.LIB
   - QDLS

- QOPT
- QFileSvr.400
- QNetWare
- QNTC

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "link()—Create Link to File" on page 153—Create Link to File
- "QlgSymlink()—Make Symbolic Link (using NLS-enabled path name)" on page 300—Make Symbolic Link (using NLS-enabled path name)
- "readlink()—Read Value of Symbolic Link" on page 452—Read Value of Symbolic Link
- "unlink()—Remove Link to File" on page 492—Remove Link to File

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **symlink()**:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

main() {
  char fn[]="readlink.file";
  char sl[]="readlink.symlink";
  char buf[30];
  int  file_descriptor;

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
    if (symlink(fn, sl) != 0)
      perror("symlink() error");
    else {
      if (readlink(sl, buf, sizeof(buf)) < 0)
        perror("readlink() error");
      else printf("readlink() returned '%s' for '%s'\n", buf, sl);

      unlink(sl);
    }
    unlink(fn);
  }
}
```

Output:

```
readlink() returned 'readlink.file' for 'readlink.symlink'
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

sysconf()—Get System Configuration Variables

Syntax

```
#include <unistd.h>

long sysconf(int name);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Yes

The **sysconf()** function returns the value of a system configuration option. The configuration option to be obtained is specified by **name**.

Parameters

name (Input) The named variable whose value is to be returned.

The value of **name** can be any one of the following symbols defined in the <**unistd.h**> header file, each corresponding to a system configuration option:

_SC_ARG_MAX (Not supported by the iSeries server). Represents ARG_MAX, which indicates the maximum number of bytes of arguments and environment data that can be passed in an **exec** function.

_SC_CHILD_MAX (Not supported by the iSeries server). Represents CHILD_MAX, which indicates the maximum number of jobs that a real user ID (UID) can have running simultaneously.

_SC_CLK_TCK Represents the CLK_TCK macro, which indicates the number of clock ticks in a second. CLK_TCK is defined in the <**time.h**> header file.

_SC_JOB_CONTROL (Not supported by the iSeries server). Represents the _POSIX_JOB_CONTROL macro, which indicates that certain job control operations are implemented by this version of the operating system. If _POSIX_JOB_CONTROL is defined (in the <**unistd.h**> header file), various APIs, such as **setpgid()**, provide more function than when the macro is not defined.

_SC_NGROUPS_MAX Represents NGROUPS_MAX, which indicates the maximum number of supplementary group IDs (GIDs) that can be associated with a job.

_SC_OPEN_MAX Represents OPEN_MAX, which indicates the maximum number of files that a single job can have open at one time.

_SC_PAGESIZE Represents the system hardware page size. The symbol _SC_PAGESIZE is defined as the decimal value 11.

_SC_PAGE_SIZE Represents the system hardware page size. The symbol _SC_PAGE_SIZE is defined as the decimal value 12.

_SC_SAVED_IDS (Not supported by the iSeries server). Represents the _POSIX_SAVED_IDS macro, which indicates that this POSIX implementation has a saved set UID and a saved set GID. If the macro exists, it is defined in the <**unistd.h**> header file. This symbol affects the behavior of such functions as **setuid()** and **setgid()**.

_SC_STREAM_MAX Represents the STREAM_MAX macro, which indicates the maximum number of streams that a job can have open at one time. The macro is defined in the <**limits.h**> header file.

_SC_TZNAME_MAX (Not supported by the iSeries server). Represents the TZNAME_MAX macro, which indicates the maximum length of the name of a time zone. If the macro exists, it is defined in the <**limits.h**> header file.

_SC_VERSION (Not supported by the iSeries server). Represents the _POSIX_VERSION macro, which indicates the version of the POSIX.1 standard that the system conforms to. If the macro exists, it is defined in the <**unistd.h**> header file.

_SC_CCSID Represents the default coded character set identifier (CCSID) used internally for integrated file system path names. A CCSID uniquely identifies the coded graphic character representation of a path name and includes such information as the character set and code page identifier. The symbol _SC_CCSID is defined as the decimal value 10.

Authorities

No authorization is required.

Return Value

value	**sysconf()** was successful. The value associated with the specified option is returned.
-1	One of the following has occurred: • The variable corresponding to **name** is valid but is not supported by the system. The **errno** global variable is not changed. • **sysconf()** failed in some other way. The **errno** is set to indicate the error.

Error Conditions

If **sysconf()** is not successful, **errno** usually indicates one of the following errors. Under some conditions, **errno** could indicate an error other than those listed here.

Error condition	Additional information
[EBADFID (page 546)]	
[EINVAL (page 540)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)

Example

See Code disclaimer information for information pertaining to code examples.

The following example determines the value of OPEN_MAX:

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

main() {
  long result;

  errno = 0;
  puts("examining OPEN_MAX limit");
  if ((result = sysconf(_SC_OPEN_MAX)) == -1)
    if (errno == 0)
      puts("OPEN_MAX is not supported.");
    else perror("sysconf() error");
  else
    printf("OPEN_MAX is %ld\n", result);
}
```

Output:

```
examining OPEN_MAX limit
OPEN_MAX is 200
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

umask()—Set Authorization Mask for Job

Syntax

```
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Yes

Every job has a file creation mask. When a job starts, the value of the file creation mask is zero. The value of zero means that no permissions are masked when a file or directory is created in the job. The **umask()** function changes the value of the file creation mask for the current job to the value specified in *cmask*.

The *cmask* argument controls file permission bits that should be set whenever the job creates a file. File permission bits set to 1 in the file creation mask are set to 0 in the file permission bits of files that are created by the job.

For example, if a call to **open()** specifies a *mode* argument with file permission bits, the file creation mask of the job affects the *mode* argument; bits that are 1 in the mask are set to 0 in the *mode* argument and, therefore, in the mode of the created file.

Only the file permission bits of *cmask* are used. The other bits in *cmask* must be cleared (not set), or the CPFA0D3 message is issued.

Parameters

cmask (Input) The new value of the file creation mask. For a description of the permission bits, see “chmod()—Change File Authorizations” on page 22.

Authorities

No authorization is required.

Return Value

umask() returns the previous value of the file creation mask. It does not return -1 or set the *errno* global variable.

Error Conditions

None.

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D3 E	cmask parameter is not valid.

Message ID	Error Message Text
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. QNTC File System Differences

   umask() does not update the file creation mask for QNTC. The settings specified in *cmask* are ignored.

Related Information

- The <**sys/stat.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "chmod()—Change File Authorizations" on page 22—Change File Authorizations
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "mkdir()—Make Directory" on page 169—Make Directory
- "open()—Open File" on page 195—Open File

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **umask()**:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>

main()
{
  int  file_descriptor;
  struct stat info;

  umask(S_IRWXG);

  if ((file_descriptor =
       creat("umask.file", S_IRWXU|S_IRWXG)) < 0)
     perror("creat() error");
  else {
    fstat(file_descriptor, &info);
    printf("permissions are: %08x\n", info.st_mode);
    close(file_descriptor);
    unlink("umask.file");
  }
}
```

Output:

```
permissions are: 000081c0
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

unlink()—Remove Link to File

Syntax

```
#include <unistd.h>

int unlink(const char *path);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 496.

The **unlink()** function removes a directory entry that refers to a file. This **unlink()** deletes the link named by *path* and decrements the link count for the file itself.

If the link count becomes zero and no job currently has the file open, the file itself is deleted. The space occupied by the file is freed for new use, and the current contents of the file are lost. If one or more jobs have the file open when the last link is removed, **unlink()** removes the link, but the file itself is not removed until the last job closes the file.

unlink() cannot be used to remove a directory; use **rmdir()** instead.

If *path* refers to a symbolic link, **unlink()** removes the symbolic link but not a file or directory named by the contents of the symbolic link.

If **unlink()** succeeds, the change and modification times for the parent directory are updated. If the link count of the file is not zero, the change time for the file is also updated. If **unlink()** fails, the link is not removed.

If the file is checked out, **unlink()** fails with the [EBUSY] error. If the file is marked "read-only", **unlink()** fails with the [EROOBJ] error.

Parameters

path (Input) A pointer to the null-terminated path name of the file to be unlinked.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgUnlink()—Remove Link to File (using NLS-enabled path name)" on page 302 for a description and an example of supplying the *path* in any CCSID.

Authorities

Note: Adopted authority is not used.

Authorization Required for unlink() (excluding QSYS.LIB, independent ASP QSYS.LIB, QDLS and QOPT)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be unlinked	*X	EACCES
Parent directory of the object to be unlinked	*WX	EACCES
Object to be unlinked	*OBJEXIST	EACCES
Parent directory of the object to be unlinked has the S_ISVTX mode bit set to binary one (see **Note**).	*ALLOBJ, or owner of the object to be unlinked, or owner of the parent directory of the object to be unlinked	EPERM

Object Referred to	Authority Required	errno
Note: The S_ISVTX mode bit (which is equivalent to the 'Restricted rename and unlink' object attribute) restriction only applies to objects in the "root" (/), QOpenSys, and user-defined file systems.		

Authorization Required for unlink() in the QSYS.LIB and independent ASP QSYS.LIB File Systems

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be unlinked	*X	EACCES
Parent directory of the object to be unlinked	See **Note**	EACCES
Object to be unlinked	See **Note**	EACCES
Note: The required authorization varies for each object type. See the DLTxxx commands in the iSeries Security Reference book for details.		

Authorization Required for unlink() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object to be unlinked	*X	EACCES
Parent directory of the object to be unlinked	*X	EACCES
Object to be unlinked	*ALL	EACCES

Authorization Required for unlink() in the QOPT File System

Object Referred to	Authority Required	errno
Volume authorization list	*CHANGE	EACCES
Each directory in the path name preceding the object to be unlinked if volume media format is Universal Disk Format (UDF)	*X	EACCES
Parent directory of the object to be unlinked if volume media format is Universal Disk Format (UDF)	*WX	EACCES
Object to be unlinked if volume media format is Universal Disk Format (UDF)	*W	EACCES
Object to be unlinked and parent directories if volume media format is not Universal Disk Format (UDF)	None	None

Return Value

0 **unlink()** was successful.

-1 **unlink()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **unlink()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	The file may be checked out.
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EDATALINK (page 547)]	
[EEXIST (page 543)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ENOTDIR (page 541)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	**unlink()** is not permitted on directories in this part of the directory hierarchy, or **unlink()** is permitted but the user does not have sufficient authority.
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	
[EXDEV (page 541)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	

Error condition	Additional information
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   The link to a file cannot be removed when a job has the file open.

   The following object types cannot be unlinked when there are secondary threads active in the job: *CFGL, *CNNL, *COSD, *CTLD, *DEVD, *IPXD, *LIND, *MODD, *NTBD, *NWID, *NWSD. The operation will fail with error code [ENOTSAFE].
3. QDLS File System Differences

   The link to a document cannot be removed when a job has the document open (returns the [EBUSY] error).
4. QOPT File System Differences

   The change and modification times of the parent directory are not updated.

   The link to a file cannot be removed when a job has the file open.

5. The link to a file cannot be removed if the file is a DataLink column in an SQL table and a row in that SQL table references this file.

Related Information

- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "close()—Close File or Socket Descriptor" on page 34—Close File or Socket Descriptor
- "link()—Create Link to File" on page 153—Create Link to File
- "open()—Open File" on page 195—Open File
- "QlgOpen()—Open a File (using NLS-enabled path name)" on page 273—Open a File (using NLS-enabled path name)
- "QlgRmdir()—Remove Directory (using NLS-enabled path name)" on page 290—Remove Directory (using NLS-enabled path name)
- "QlgUnlink()—Remove Link to File (using NLS-enabled path name)" on page 302—Remove Link to File (using NLS-enabled path name)
- "rmdir()—Remove Directory" on page 463—Remove Directory

Example

See Code disclaimer information for information pertaining to code examples.

The following example removes a link to a file:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

main() {
  int file_descriptor;
  char fn[]="unlink.file";

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
    if (unlink(fn) != 0)
      perror("unlink() error");
  }
}
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

utime()—Set File Access and Modification Times

Syntax

```
#include <utime.h>

int utime(const char *path, const struct utimbuf *times);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 500.

The **utime()** function sets the access and modification times of *path* to the values in the `utimbuf` structure. If *times* is a NULL pointer, the access and modification times are set to the current time. If the named file is a symbolic link, **utime()** resolves the symbolic link.

If the file is checked out by another user (someone other than the user profile of the current job), **utime()** fails with the [EBUSY] error.

When **utime()** completes successfully, it marks the change time of the file to be updated.

Parameters

path
(Input) A pointer to the null-terminated path name of the file for which the times should be changed.

This parameter is assumed to be represented in the CCSID (coded character set identifier) currently in effect for the job. If the CCSID of the job is 65535, this parameter is assumed to be represented in the default CCSID of the job.

See "QlgUtime()—Set File Access and Modification Times (using NLS-enabled path name)" on page 303 for a description and an example of supplying the *path* in any CCSID.

times
(Input) A pointer to a structure `utimbuf`, which contains the times to be updated.

The structure `utimbuf` is defined according to the POSIX.1 definition as follows:

```
struct utimbuf {
   time_t   actime;   /* The new access time       */
   time_t   modtime;  /* The new modification time */
   }
```

The `time_t` type gives the number of seconds since the *Epoch*.

Authorities

Note: Adopted authority is not used.

Authorization Required for utime() (excluding QDLS)

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object when changing the time to a specified value	Owner (See **Note**)	EPERM
Object when changing the time to the current time	Owner or *W (See **Note**)	EACCES
Note: You do not need the listed authority if you have *ALLOBJ special authority.		

Authorization Required for utime() in the QDLS File System

Object Referred to	Authority Required	errno
Each directory in the path name preceding the object	*X	EACCES
Object when changing the time to a specified value	*W	EPERM
Object when changing the time to the current time	*W	EACCES

Return Value

0
utime() was successful. The file access and modification times are changed.

-1 **utime()** was not successful. The file times are not changed. The *errno* global variable is set to indicate the error.

Error Conditions

If **utime()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems. *times* is NULL and the job does not have authority to perform the requested function.
[EAGAIN (page 541)]	
[EBADFID (page 546)]	
[EBADNAME (page 540)]	
[EBUSY (page 540)]	
[ECONVERT (page 545)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFILECVT (page 546)]	
[EINTR (page 541)]	
[EINVAL (page 540)]	
[EIO (page 540)]	
[EISDIR (page 544)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ELOOP (page 544)]	
[ENAMETOOLONG (page 544)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOENT (page 540)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTDIR (page 541)]	
[ENOTSAFE (page 546)]	
[ENOTSUP (page 542)]	
[EPERM (page 540)]	*times* is not NULL and the thread does not have authority to perform the requested function.
[EROOBJ (page 545)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA0D4 E	File system error occurred. Error number &1.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - ″Root″ (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   These file systems do not support **utime()**.
3. QDLS File System Differences

   Changing the times of the /QDLS directory (the root folder) is not allowed.
4. QOPT File System Differences

   The QOPT file system does not support **utime()**.

5. QNTC File System Differences

   The QNTC file system does not set the access and modification times of *path*. The values in the `utimbuf` structure are ignored.

Related Information

- The <**utime.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**limits.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "QlgUtime()—Set File Access and Modification Times (using NLS-enabled path name)" on page 303—Set File Access and Modification Times (using NLS-enabled path name)

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **utime()**:

```
#include <utime.h>
#include <time.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main() {
  int file_descriptor;
  char fn[]="utime.file";
  struct utimbuf ubuf;
  struct stat info;

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(file_descriptor);
    puts("before utime()");
    stat(fn,&info);
    printf("  utime.file modification time is %ld\n",
           info.st_mtime);
    ubuf.modtime = 0;       /* set modification time to Epoch */
    time(&ubuf.actime);
    if (utime(fn, &ubuf) != 0)
      perror("utime() error");
    else {
      puts("after utime()");
      stat(fn,&info);
      printf("  utime.file modification time is %ld\n",
             info.st_mtime);
    }
    unlink(fn);
  }
}
```

Output:

```
before utime()
  utime.file modification time is 749323571
after utime()
  utime.file modification time is 0
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

write()—Write to Descriptor

Syntax

```
#include <unistd.h>

ssize_t write
  (int file_descriptor, const void *buf, size_t nbyte);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 506.

The **write()** function writes *nbyte* bytes from *buf* to the file or socket associated with *file_descriptor*. *nbyte* should not be greater than INT_MAX (defined in the <**limits.h**> header file). If *nbyte* is zero, **write()** simply returns a value of zero without attempting any other action.

If *file_descriptor* refers to a "regular file" (a stream file that can support positioning the file offset) or any other type of file on which the job can do an **lseek()** operation, **write()** begins writing at the file offset associated with *file_descriptor*, unless O_APPEND is set for the file (see below). A successful **write()** increments the file offset by the number of bytes written. If the incremented file offset is greater than the previous length of the file, the length of the file is set to the new file offset.

If O_APPEND (defined in the <**fcntl.h**> header file) is set for the file, **write()** sets the file offset to the end of the file before writing the output.

If there is not enough room to write the requested number of bytes (for example, because there is not enough room on the disk), the **write()** function writes as many bytes as the remaining space can hold.

If **write()** is successful and *nbyte* is greater than zero, the change and modification times for the file are updated.

If *file_descriptor* refers to a descriptor obtained using the **open()** function with O_TEXTDATA specified, the data is written to the file assuming it is in textual form. The maximum number of bytes on a single write that can be supported for text data is 2,147,483,408 (2GB - 240) bytes. The data is converted from the code page of the application, job, or system to the code page of the file as follows:

- When writing to a true stream file, any line-formatting characters (such as carriage return, tab, and end-of-file) are just converted from one code page to another.
- When writing to a record file that is being used as a stream file:
  - End-of-line characters are removed.
  - Records are padded with blanks (for a source physical file member) or nulls (for a data physical file member).
  - Tab characters are replaced by the appropriate number of blanks to the next tab position.

There are some important considerations if O_CCSID was specified on the **open()**.

- The **write()** will attempt to convert all of the data in the user's buffer. Successfully converted data will be written. Unconverted data is usually assumed to be a partial character. Partial characters will be buffered internally and data from the next consecutive write will be appended to the buffered data. If incorrect data is provided on a consecutive write, the write may fail with the [ECONVERT] error.

  If an **lseek()** is performed, the file is closed, or the current job is ended, the buffered data will be discarded. Discarded data will not be written to the file. See "lseek()—Set File Read/Write Offset" on page 157—Set File Read/Write Offset for more information.
- Because of the above consideration and because of the possible expansion or contraction of converted data, applications using the O_CCSID flag should avoid assumptions about data size and the current

file offset. For example, the user may supply a buffer to 100 bytes, but after an application has written the buffer to a new file, the file size may be 50, 200, or something else, depending on the CCSIDs involved.

If O_TEXTDATA was not specified on the **open()**, the data is written to the file without conversion. The application is responsible for handling the data.

When *file_descriptor* refers to a socket, the **write()** function writes to the socket identified by the socket descriptor.

Note: When the write completes successfully, the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode will be cleared. If the write is unsuccessful, the bits are undefined.

Write requests to a pipe or FIFO are handled the same as a regular file, with the following exceptions:
- The S_ISUID and S_ISGID file mode bits will not be cleared.
- There is no file offset associated with a pipe or FIFO. Each write request will append to the end of the pipe or FIFO.
- Write requests of [PIPE_BUF] bytes or less will not be interleaved with data from other threads performing writes on the same pipe or FIFO. Writes of greater than [PIPE_BUF] bytes may have data interleaved on arbitrary boundaries with writes by other threads, whether or not the O_NONBLOCK flag of the file status flags is set.
- If the O_NONBLOCK flag was not specified and the pipe or FIFO is full, the write request will block the calling thread until the requested amount of data in *nbyte* is written.
- If the O_NONBLOCK flag was specified, then the following pertain to various write requests:
  - The **write()** function will not block the calling thread.
  - A write request for [PIPE_BUF] or fewer bytes will have the following effect:

    If there is sufficient space available in the pipe or FIFO, **write()** will transfer all the data and return the number of bytes requested. If there is not sufficient space in the pipe or FIFO, **write()** will transfer no data, return -1, and set *errno* to [EAGAIN].
  - A write request for more than [PIPE_BUF] bytes will cause one of the following:
    - When at least one byte can be written, **write()** will transfer what it can and return the number of bytes written.
    - When no data can be written, **write()** will transfer no data, return -1, and set *errno* to [EAGAIN].

Parameters

file_descriptor
(Input) The descriptor of the file to which the data is to be written.

buf (Input) A pointer to a buffer containing the data to be written.

nbyte (Input) The size in bytes of the data to be written.

Authorities

No authorization is required.

Return Value

value **write()** was successful. The value returned is the number of bytes actually written. This number is less than or equal to *nbyte*.

-1 **write()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **write()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems. If writing to a socket, this error code indicates one of the following: • The destination address specified is a broadcast address and the socket option `SO_BROADCAST` was not set (with a *setsockopt()*). • The process does not have the appropriate privileges to the destination address. This error code can only be returned on a socket with an address family of `AF_INET` and a type of `SOCK_DGRAM`.
[EAGAIN (page 541)]	If *file_descriptor* refers to a pipe or FIFO that has its O_NONBLOCK flag set, this error occurs if the **write()** would have blocked the calling thread.
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFBIG (page 545)]	The size of the object would exceed the system allowed maximum size or the process soft file size limit. The file is a regular file, *nbyte* is greater than 0, and the starting offset is greater than or equal to 2 GB minus 2 bytes.
[EINTR (page 541)]	
[EINVAL (page 540)]	For example, the file system that the file resides in does not support large files, and the starting offset exceeds 2GB minus 2 bytes.
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ENXIO (page 541)]	
[ERESTART (page 547)]	
[ETRUNC (page 540)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN (page 544)]	

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

Error condition	Additional information
[ECONNREFUSED (page 542)]	This error code can only be returned on sockets that use a connectionless transport service.

Error condition	Additional information
[EDESTADDRREQ (page 542)]	A destination address has not been associated with the socket pointed to by the *fildes* parameter. This error code can only be returned on sockets that use a connectionless transport service.
[EHOSTDOWN (page 542)]	This error code can only be returned on sockets that use a connectionless transport service.
[EHOSTUNREACH (page 542)]	This error code can only be returned on sockets that use a connectionless transport service.
[EINTR (page 541)]	
[EMSGSIZE (page 542)]	The data to be sent could not be sent atomically because the size specified by *nbyte* is too large.
[ENETDOWN (page 542)]	This error code can only be returned on sockets that use a connectionless transport service.
[ENETUNREACH (page 542)]	This error code can only be returned on sockets that use a connectionless transport service.
[ENOBUFS (page 542)]	
[ENOTCONN (page 542)]	This error code is returned only on sockets that use a connection-oriented transport service.
[EPIPE (page 543)]	
[EUNATCH (page 543)]	
[EWOULDBLOCK (page 541)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. QSYS.LIB and independent ASP QSYS.LIB File System Differences

   This function will fail with error code [ENOTSAFE] if the object on which this function is operating is a save file and multiple threads exist in the job.

   If the file specified is a save file, only complete records will be written into the save file. A **write()** request that does not provide enough data to completely fill a save file record will cause the partial record's data to be saved by the file system. The saved partial record will then be combined with additional data on subsequent **write()**'s until a complete record may be written into the save file. If the save file is closed prior to a saved partial record being written into the save file, then the saved partial record is discarded, and the data in that partial record will need to be written again by the application.

   A successful **write()** updates the change, modification, and access times for a database member using the normal rules that apply to database files. At most, the access time is updated once per day.

   You should be careful when writing end-of-file characters in the QSYS.LIB and independent ASP QSYS.LIB file systems. These file systems end-of-file characters are symbolic; that is, they are stored outside the file member. However, some situations can result in actual, nonsymbolic end-of-file characters being written to a member. These nonsymbolic end-of-file characters could cause some tools or utilities to fail. For example:
   - If you previously wrote an end-of-file character as the last character of a member, do not continue to write data after that end-of-file character. Continuing to write data will cause a nonsymbolic end-of-file to be written. As a result, a compile of the member could fail.
   - If you previously wrote an end-of-file character as the last character of a member, do not write other end-of-file characters preceding it in the file. This will cause a nonsymbolic end-of-file to be written. As a result, a compile of the member could fail.
   - If you previously used the integrated file system interface to manipulate a member that contains an end-of-file character, avoid using other interfaces (such as the Source Entry Utility or database reads and writes) to manipulate the member. If you use other interfaces after using the integrated file system interface, the end-of-file information will be lost.
3. QOPT File System Differences

   The change and modification times of the file are updated when the file is closed.

   When writing to files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being written are ignored.
4. Network File System Differences

Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations (several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data).

Reading and writing to files with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the **fcntl()** API to get and release these locks.

5. QFileSvr.400 File System Differences

   The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.

6. Sockets Usage Notes

   a. **write()** only works with sockets on which a **connect()** has been issued, since it does not allow the caller to specify a destination address.
   b. To broadcast on an `AF_INET` socket, the socket option `SO_BROADCAST` must be set (with a **setsockopt()**).
   c. When using a connection-oriented transport service, all errors except `[EUNATCH]` and `[EUNKNOWN]` are mapped to `[EPIPE]` on an output operation when either of the following occurs:

      - A connection that is in progress is unsuccessful.
      - An established connection is broken.

      To get the actual error, use **getsockopt()** with the `SO_ERROR` option, or perform an input operation (for example, **read()**).

7. For the file systems that do not support large files, **write()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **write()** will return [EFBIG] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.
8. Using this function successfully on the /dev/null or /dev/zero character special file results in a return value of the total number of bytes requested to be written. No data is written to the character special file. In addition, the change and modification times for the file are updated.
9. If the write exceeds the process soft file size limit, signal SIFXFSZ is issued.

Related Information

- The <**fcntl.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- The <**unistd.h**> file (see "Header Files for UNIX-Type Functions" on page 537)
- "creat()—Create or Rewrite File" on page 40—Create or Rewrite File
- "dup()—Duplicate Open File Descriptor" on page 55—Duplicate Open File Descriptor
- "dup2()—Duplicate Open File Descriptor to Another Descriptor" on page 58—Duplicate Open File Descriptor to Another Descriptor
- "fclear()—Write (Binary Zeros) to Descriptor" on page 77—Write (Binary Zeros) to Descriptor
- "fclear64()—Write (Binary Zeros) to Descriptor (Large File Enabled)" on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)
- "fcntl()—Perform File Control Command" on page 82—Perform File Control Command
- "ioctl()—Perform I/O Control Request" on page 141—Perform I/O Control Request
- "lseek()—Set File Read/Write Offset" on page 157—Set File Read/Write Offset

- "open()—Open File" on page 195—Open File
- "pread()—Read from Descriptor with Offset" on page 223—Read from Descriptor with Offset
- "pread64()—Read from Descriptor with Offset (large file enabled)" on page 228—Read from Descriptor with Offset (large file enabled)
- "pwrite()—Write to Descriptor with Offset" on page 229—Write to Descriptor with Offset
- "pwrite64()—Write to Descriptor with Offset (large file enabled)" on page 234—Write to Descriptor with Offset (large file enabled)
- "read()—Read from Descriptor" on page 437—Read from Descriptor
- "readv()—Read from Descriptor Using Multiple Buffers" on page 455—Read from Descriptor Using Multiple Buffers
- send()—Send Data
- sendmsg()—Send Data or Descriptors or Both
- sendto()—Send Data
- "writev()—Write to Descriptor Using Multiple Buffers" on page 509—Write to Descriptor Using Multiple Buffers

Example

See Code disclaimer information for information pertaining to code examples.

The following example writes a specific number of bytes to a file:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define mega_string_len 1000000

main() {
  char *mega_string;
  int  file_descriptor;
  int  ret;
  char fn[]="write.file";

  if ((mega_string = (char*) malloc(mega_string_len)) == NULL)
    perror("malloc() error");
  else if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    memset(mega_string, '0', mega_string_len);
    if ((ret = write(file_descriptor, mega_string, mega_string_len)) == -1)
      perror("write() error");
    else printf("write() wrote %d bytes\n", ret);
    if (close(file_descriptor)!= 0)
       perror("close() error");
    if (unlink(fn)!= 0)
       perror("unlink() error");
  }
  free(mega_string);
}
```

Output:

```
write() wrote 1000000 bytes
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

writev()—Write to Descriptor Using Multiple Buffers

Syntax

```
#include <sys/types.h>
#include <sys/uio.h>

int writev(int descriptor,
           struct iovec *io_vector[],
           int vector_length)
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 511.

The **writev()** function is used to write data to a file or socket descriptor. **writev()** provides a way for the data that is going to be written to be stored in several different buffers (*scatter/gather I/O*).

Note: When the write completes successfully, the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode will be cleared. If the write is unsuccessful, the bits are undefined.

See "write()—Write to Descriptor" on page 502 for more information related to writing to a descriptor.

Parameters

descriptor
(Input) The descriptor to which the data is to be written. The descriptor refers to either a file or a socket.

io_vector[]
(Input) The pointer to an array of type **struct iovec**. **struct iovec** contains a sequence of pointers to buffers in which the data to be written is stored. The structure pointed to by the *io_vector* parameter is defined in **<sys/uio.h>**.

```
struct iovec {
   void      *iov_base;
   size_t   iov_len;
}
```

iov_base and *iov_len* are the only fields in *iovec* used by sockets. *iov_base* contains the pointer to a buffer and *iov_len* contains the buffer length. The rest of the fields are reserved.

vector_length
(Input) The number of entries in *io_vector*.

Authorities

No authorization is required.

Return Value

value	**writev()** was successful. The value returned is the number of bytes actually written.
-1	**writev()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **writev()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES (page 541)]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems. If writing to a socket, this error code indicates one of the following: • The destination address specified is a broadcast address and the socket option `SO_BROADCAST` was not set (with a *setsockopt()*). • The process does not have the appropriate privileges to the destination address. This error code can only be returned on a socket with an address family of `AF_INET` and a type of `SOCK_DGRAM`.
[EAGAIN (page 541)]	
[EBADF (page 543)]	
[EBADFID (page 546)]	
[EBUSY (page 540)]	
[EDAMAGE (page 544)]	
[EFAULT (page 541)]	
[EFBIG (page 545)]	The size of the object would exceed the system allowed maximum size or the process soft file size limit. The file is a regular file, *nbyte* is greater than 0, and the starting offset is greater than or equal to 2 GB minus 2 bytes.
[EINTR (page 541)]	
[EINVAL (page 540)]	For example, the file resides in a file system that does not support large files, and the starting offset exceeds 2GB minus 2 bytes.
[EIO (page 540)]	
[EJRNDAMAGE (page 546)]	
[EJRNENTTOOLONG (page 547)]	
[EJRNINACTIVE (page 546)]	
[EJRNRCVSPC (page 547)]	
[ENEWJRN (page 547)]	
[ENEWJRNRCV (page 547)]	
[ENOMEM (page 543)]	
[ENOSPC (page 541)]	
[ENOTAVAIL (page 547)]	
[ENOTSAFE (page 546)]	
[ERESTART (page 547)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETRUNC (page 540)]	
[EUNKNOWN (page 544)]	

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

Error condition	Additional information
[ECONNREFUSED (page 542)]	This error code can only be returned on sockets that use a connectionless transport service.
[EDESTADDRREQ (page 542)]	A destination address has not been associated with the socket pointed to by the *fildes* parameter. This error code can only be returned on sockets that use a connectionless transport service.
[EHOSTDOWN (page 542)]	This error code can only be returned on sockets that use a connectionless transport service.
[EHOSTUNREACH (page 542)]	This error code can only be returned on sockets that use a connectionless transport service.

Error condition	Additional information
[EINTR (page 541)]	
[EMSGSIZE (page 542)]	The data to be sent could not be sent atomically because the size specified by *nbyte* is too large.
[ENETDOWN (page 542)]	This error code can only be returned on sockets that use a connectionless transport service.
[ENETUNREACH (page 542)]	This error code can only be returned on sockets that use a connectionless transport service.
[ENOBUFS (page 542)]	
[ENOTCONN (page 542)]	This error code is returned only on sockets that use a connection-oriented transport service.
[EPIPE (page 543)]	
[EUNATCH (page 543)]	
[EWOULDBLOCK (page 541)]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL (page 541)]	
[ECONNABORTED (page 542)]	
[ECONNREFUSED (page 542)]	
[ECONNRESET (page 542)]	
[EHOSTDOWN (page 542)]	
[EHOSTUNREACH (page 542)]	
[ENETDOWN (page 542)]	
[ENETRESET (page 542)]	
[ENETUNREACH (page 542)]	
[ESTALE (page 546)]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT (page 543)]	
[EUNATCH (page 543)]	

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

- ″Root″ (/)
- QOpenSys
- User-defined
- QNTC
- QSYS.LIB
- Independent ASP QSYS.LIB
- QOPT
- Network File System
- QFileSvr.400

2. **writev()** only works with sockets on which a *connect()* has been issued, since the call does not allow the caller to specify a destination address.
3. **writev()** is an atomic operation on sockets of type `SOCK_DGRAM` and `SOCK_RAW` in that it produces one packet of data every time it is issued. For example, a **writev()** to a datagram socket results in a single datagram.
4. To broadcast on an `AF_INET` socket, the socket option `SO_BROADCAST` must be set (with a *setsockopt()*).
5. When using a connection-oriented transport service, all errors except `[EUNATCH]` and `[EUNKNOWN]` are mapped to `[EPIPE]` on an output operation when either of the following occurs:
   - A connection that is in progress is unsuccessful.
   - An established connection is broken.

   To get the actual error, use *getsockopt()* with the `SO_ERROR` option, or perform an input operation (for example, *read()*).
6. For the file systems that do not support large files, **writev()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **writev()** will return [EFBIG] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.
7. QFileSvr.400 File System Differences

   The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.
8. QOPT File System Differences

   When writing to files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being written are ignored.
9. Using this function successfully on the dev/null or /dev/zero character special file results in a return value of the total number of bytes requested to be written. No data is written to the character special file. In addition, the change and modification times for the file are updated.
10. If the write exceeds the process soft file size limit, signal SIFXFSZ is issued.

Related Information

- The <**fcntl.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- The <**unistd.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- “creat()—Create or Rewrite File” on page 40—Create or Rewrite File
- “dup()—Duplicate Open File Descriptor” on page 55—Duplicate Open File Descriptor
- “dup2()—Duplicate Open File Descriptor to Another Descriptor” on page 58—Duplicate Open File Descriptor to Another Descriptor
- “fclear()—Write (Binary Zeros) to Descriptor” on page 77—Write (Binary Zeros) to Descriptor
- “fclear64()—-Write (Binary Zeros) to Descriptor (Large File Enabled)” on page 82—Write (Binary Zeros) to Descriptor (Large File Enabled)
- “fcntl()—Perform File Control Command” on page 82—Perform File Control Command
- “ioctl()—Perform I/O Control Request” on page 141—Perform I/O Control Request

- "lseek()—Set File Read/Write Offset" on page 157—Set File Read/Write Offset
- "open()—Open File" on page 195—Open File
- "read()—Read from Descriptor" on page 437—Read from Descriptor
- "readv()—Read from Descriptor Using Multiple Buffers" on page 455—Read from Descriptor Using Multiple Buffers
- send()—Send Data
- sendmsg()—Send Data or Descriptors or Both
- sendto()—Send Data
- "write()—Write to Descriptor" on page 502—Write to Descriptor

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

Exit Programs

These are the Exit Programs for this category.

Integrated File System Scan on Close Exit Program

Required Parameter Group:

1	Integrated file system close exit information	Input	Char(*)
2	Status information	Output	Char(*)

QSYSINC Member Name: QP0LSCAN
Exit Point Name: QIBM_QP0L_SCAN_CLOSE
Exit Point Format Name: SCCL0100

The integrated file system scan on close exit program is called to do scan processing when an integrated file system object is closed under the following conditions.

The exit program will **not** be called if:

- No exit programs exist for this exit point.
- -or- the Scan file systems (QSCANFS) system value has *NONE specified so that no file systems will be scanned.
- -or- the object was marked to not be scanned and a scan is not required because the object was restored. (See **Note** 2)
- -or- the object being closed was opened for write access only.
- -or- the object is being used as a network server storage space or as a virtual volume. From the perspective of the server, these objects appear as byte stream files within the integrated file system.
- -or- the object is not being accessed from a file server, and the Scan file systems control (QSCANFSCTL) system value has *FSVRONLY specified so that only file server accesses are scanned. (See **Note** 2)
- -or- the object is in a file system that has not completely converted to the *TYPE2 directory format. For information on the *TYPE2 directory format, see the Convert Directory (CVTDIR) command and the Integrated file system information in the Files and file systems topic.

If the previous conditions have been met, the exit program will be called if:

- The object has never been scanned.
- -or- the object's data has been modified since the last time it was scanned. Data modifications include writes, memory map writes, truncates or clears.

- -or- the CCSID of the object has been modified since the last time it was scanned.
- -or- the To CCSID specified on the open request associated with this close is different than the last two To CCSIDs that were specified and previously scanned for this object.
- -or- the object was opened in binary in association with this close request, and it has not previously been scanned in binary.
- -or- there have been updates to the scanning software and the object was not marked to be scanned only if the object changed. (See **Note** 2) Updates to scanning software occur by either registering additional exit programs for the scan-related exit points, or by calling Change Scan Signature (QP0LCHSG) API to update the scan key signature associated with existing exit program scan keys.

Note: If there are multiple descriptors referencing the same open instance of the object, then the exit program will **only** be called for the close request on the last descriptor. Additionally, the From CCSID of the object will be the value it is at the point in time of the close operation while the To CCSID will be reflective of the value specified at open.

For more information on close processing, see "close()—Close File or Socket Descriptor" on page 34. For more information on the scan-related attributes which can be set for objects, see "Qp0lSetAttr()—Set Attributes" on page 403 For more information on the integrated file system scan processing and various options, see the Integrated file system information in the Files and file systems topic.

The exit point supports a maximum of 50 exit programs. For information about adding an exit program to an exit point, see the Registration Facility.

Notes:

1. If the integrated file system exit program returns any error messages or if any errors are received when attempting to call the exit program, the object will be treated as if the program was not called and the object was not scanned. Therefore, the close operation will continue unless the Scan file systems control (QSCANFSCTL) system value has *ERRFAIL specified which will cause the operation to fail. If a scan detects a failure, the close operation will still proceed and complete to release the resources. If the Scan file systems control (QSCANFSCTL) system value has *NOFAILCLO specified, the close operation will not return any failure indication. If *NOFAILCLO is not specified, the close operation will fail with error code [ESCANFAILURE].
2. If the oflags specified when the object being closed was opened include the O_FORCE_SCAN value, then one or more of the following conditions will be ignored when determining whether the integrated file system scan-related exit programs will be called:
   - The QSCANFSCTL system value specification of *FSVRONLY.
   - The object was marked to not be scanned (e.g. scan attribute of *NO).
   - The object was marked to be scanned only if the object changed (e.g. scan attribute of *CHGONLY).

   For example, an object is closed that has a scan attribute of *YES, and the close request is not through the file servers when *FSVRONLY is specified. If O_FORCE_SCAN is specified on the open request associated with this close request, the object will be scanned if all the remaining conditions are met. Similarly, if an object that has a scan attribute of *NO or *CHGONLY is closed whose associated open request had O_FORCE_SCAN specified, the object will be scanned if all the remaining conditions are met. See "Using the oflag Parameter" on page 197 in "open()—Open File" on page 195 for more information on oflags.

Restrictions

- Only objects of type *STMF that are in the "root" (/), QOpenSys, and user-defined file systems that have completely converted to the *TYPE2 directory format are scanned. For information on *TYPE2 directories, see the Convert Directory(CVTDIR) command and the Integrated file system information in the Files and file systems topic.
- The exit programs will not be called during an IPL or the vary-on of an independent Auxiliary Storage Pool (ASP).

- The exit programs will not be called when objects are being closed as a part of a process end request.
- During the call to the exit programs, the ASP group associated with the thread will not be able to be changed.
- The exit programs must exist in the system ASP or in a basic user ASP. They cannot exist in an independent ASP. Any ASP group could be associated with the thread when the exit program is called. If the exit program is not found, the object will be treated as if the program was not called and the object was not scanned. Therefore, the close operation will continue unless the Scan file systems control (QSCANFSCTL) system value has *ERRFAIL specified which will cause the operation to fail. If the Scan file systems control (QSCANFSCTL) system value has *NOFAILCLO specified, the close operation will not return any failure indication. If *NOFAILCLO is not specified, the close operation will fail with error code [ESCANFAILURE].
- The exit programs could be called from an exit point within a multi-threaded job and must be written to be threadsafe.

Authorities and Locks

User Profile Authority

*ALLOBJ (all object) and *SECADM (security administrator) special authorities to add exit programs to the registration facility

*ALLOBJ and *SECADM special authorities to remove exit programs from the registration facility

Program Data

When you register the exit program, the following program data must be provided. The following table shows the structure of the program data information. For a description of the fields in this format, see "Field Descriptions" on page 516. This structure is defined in header file qp0lscan.h as data type Qp0l_Scan_Program_Data_t.

Offset		Type	Field
Dec	Hex		
0	0	Char(10)	User profile
10	A	Char(20)	Scan key
30	1E	Char(12)	Scan key signature

Required Parameter Group

Integrated file system close exit information

INPUT; CHAR(*)

Information that is needed by the exit program to do its object scan processing. For details, see "Format of Integrated File System Close Exit Information (Input)."

Status information

OUTPUT; CHAR(*)

Information that is returned by the exit program indicating what scan processing has occurred. For details, see "Format of Status Information (Output)" on page 516.

Format of Integrated File System Close Exit Information (Input)

The following table shows the structure of the integrated file system close exit information for exit point format SCCL0100. For a description of the fields in this format, see "Field Descriptions" on page 516. This structure is defined in header file qp0lscan.h as data type Qp0l_Scan_Exit_Information_t.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Integrated file system close exit information length
4	4	CHAR(20)	Exit point name
24	18	CHAR(8)	Exit point format name
32	20	BINARY(4)	Length of status information
32	20	BINARY(4)	Scan descriptor
36	24	BINARY(4), UNSIGNED	From CCSID
40	28	BINARY(4), UNSIGNED	To CCSID
44	2C	BINARY(4), UNSIGNED	Last failure CCSID
48	30	BINARY(4)	Oflags
52	34	CHAR(16)	File ID
68	44	CHAR(10)	Object type
78	4E	CHAR(1)	File system
79	4F	CHAR(1)	Additional call
80	50	CHAR(1)	Object modified since last scan
81	51	CHAR(1)	Scan signatures different
82	52	CHAR(1)	Call after previous failure

Format of Status Information (Output)

The following table shows the structure of the status information. For a description of the fields in this format, see "Field Descriptions." This structure is defined in header file qp0lscan.h as data type Qp0l_Scan_Status_Information_t.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Close scan status information length
4	4	BINARY(4), UNSIGNED	Failing CCSID
8	8	CHAR(1)	Update object scan information
9	9	CHAR(1)	Scan status

Field Descriptions

Additional call. Whether the exit program was called an additional time because another "Integrated File System Scan on Close Exit Program" on page 513 that was called has indicated the object was modified. See the *scan status* field for this modify indication. The possible values are:

QP0L_SCAN_CALL_FIRST (x'00')	The first call to the exit program.
QP0L_SCAN_CALL_ADDL (x'01')	An additional call to the exit program because another exit program has indicated the object was modified.

Call after previous failure. Whether the exit program was called after the object had previously been scanned and a failure detected. The possible values are:

QP0L_SCAN_NO (x'00')	This is not a call after a previous scan failure.

QP0L_SCAN_YES (x'01') This is a call after a previous scan failure. The *Last failure CCSID* field in conjunction with the *From CCSID* indicate the CCSID or binary indication of the failing scan request.

Note: If the *Failing CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

Close scan status information length. The length in bytes of all data returned from the integrated file system close exit program. The only valid value for this field is 10. If anything else is specified, the object will be treated as if the program was not called and the object was not scanned. Therefore, the close operation will continue unless the Scan file systems control (QSCANFSCTL) system value has *ERRFAIL specified which will cause the operation to fail. If the Scan file systems control (QSCANFSCTL) system value has *NOFAILCLO specified, the close operation will not return any failure indication. If *NOFAILCLO is not specified, the close operation will fail with error code [ESCANFAILURE].

Exit point format name. The format name for the integrated file system scan on close exit program. The possible format name follows:

SCCL0100 The format name that is used while an object is being closed.

Exit point name. The name of the exit point that is calling the exit program.

Failing CCSID. This field only has meaning if the *Call after previous failure* field had a value of QP0L_SCAN_YES when the exit program was called, and if the *Update object scan information* field has a value of QP0L_SCAN_YES, and if the *Scan status* field has a value of QP0L_SCAN_FAILURE or QP0L_SCAN_FAIL_WANT_MODIFY. When the *Call after previous failure* had a value of QP0L_SCAN_YES, then the scan exit program should verify that the object does not have any problems when scanned using both the *To CCSID* and *Last failure CCSID* values. If either scan fails, then this field should be filled in with the failing CCSID which will be stored as part of the object scan information with the failure indication. If the value of this field does not match either of the two input CCSID fields, then the *To CCSID* value will be used. If more than one exit program indicates a failure, the failing CCSID value which will be preserved is from the last exit program which scanned the object and indicated a failure. For more information on CCSIDs and conversions, see "Coded Character Set Identifier (CCSID) Information" on page 531 in "Integrated File System Scan on Open Exit Program" on page 523.

Note: If the *Failing CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

File ID. A unique identifier associated with the object that is being closed. A file ID can be used with "Qp0lGetPathFromFileID()—Get Path Name of Object from Its File ID" on page 351, to retrieve an object's path name.

File system. The file system that the object being scanned is in. The possible value follows:

QP0L_SCAN_ROOT_QOPENSYS_UDFS (x'00') The object is in the "root" (/), QOpenSys, or a user-defined file system.

From CCSID. The CCSID value that the data is in on the system itself at the point in time of the close operation. Therefore, this will be the CCSID in which data is to be returned (when reading from the object using the *Scan descriptor*), or the CCSID in which data is being supplied (when writing to the object using the *Scan descriptor*). For more information on CCSIDs and conversions, see "Coded Character Set Identifier (CCSID) Information" on page 531 in "Integrated File System Scan on Open Exit Program" on page 523.

Integrated file system close exit information length. The length in bytes of all data passed to the integrated file system close exit program.

Last failure CCSID. The CCSID value that was specified when this object was last scanned and indicated a scan failure. This field only has meaning if the *Call after previous failure* field has a value of QP0L_SCAN_YES. Therefore, this would have been the CCSID in which data was to have been returned (when the user was to be reading from the object), or the CCSID in which data was to have been supplied (when the user was to be writing to the object). However, that request failed for this CCSID. This is now being returned so that this CCSID can also be scanned, if it is different than the *To CCSID* value. For more information on CCSIDs and conversions, see "Coded Character Set Identifier (CCSID) Information" on page 531 in "Integrated File System Scan on Open Exit Program" on page 523.

Note: If the *Last failure CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

Length of status information. The length in bytes allocated for the returned status information.

Object modified since last scan. Whether the exit program was called because the objects data or CCSID has been modified since it was last scanned. Examples of object data or CCSID modifications are: writing to the object, directly or through memory mapping; truncating the object; clearing the object; and changing the objects CCSID attribute, etc.. The possible values are:

QP0L_SCAN_NO (x'00')	The object has not been modified since it was last scanned.
QP0L_SCAN_YES (x'01')	The object has been modified since it was last scanned.

Object type. The object type. See Control Language (CL) information in the iSeries Information Center for descriptions of all object types.

Oflags. The oflags that were specified on the open request associated with this close request with the following exceptions. For a description of all possible oflag values, see "open()—Open File" on page 195

- If the oflags do not contain write access, the system will attempt to upgrade the access intent to include write, unless the Scan file systems control (QSCANFSCTL) system value has *NOWRTUPG specified or the object is not eligible for write access. If the upgrade is not attempted or is unsuccessful, the access intent matches the users invocation. If it is successful, the write access intent is included in this oflag information. This upgrade would be useful if the exit program wanted to modify the object to correct any problems found while scanning.
- The CCSID related flags will have been removed. This includes O_TEXTDATA, O_CCSID, O_CODEPAGE, and O_TEXT_CREATE.
- The synchronization flags will have been removed. This includes O_SYNC, O_DSYNC, and O_RSYNC.

Scan descriptor. A descriptor representing the object that is being closed. This scan descriptor has the following characteristics:

- It can be used to do any read processing on the object being processed. Reads using this descriptor will not update the last access timestamp information for the object.
- It can be used to do any write processing on the object being processed. If write processing is done by the exit program, the exit program should indicate QP0L_SCAN_MODIFY in the *Scan status* field. If it does not, the object's scan information will be cleared as if the objects data has been modified.
- It cannot be used to memory map the object, see "mmap()—Memory Map a File" on page 179
- It cannot be used to close the object using "close()—Close File or Socket Descriptor" on page 34. When control returns from the exit program, the system code will do the close of this *scan descriptor*. The system will wait on this close attempt until all accesses to this object are closed. Therefore, if the exit program uses givedescriptor()—Pass Descriptor Access to Another Job and takedescriptor()—Receive Socket Access from Another Job or sendmsg()—Send Data or Descriptors or Both and

recvmsg()—Receive Data or Descriptors or Both to pass the descriptor to another job, the job which used takedescriptor() or recvmsg() must close that descriptor when it is done processing, else the system will be waiting for that close.

- “dup()—Duplicate Open File Descriptor” on page 55 and “fcntl()—Perform File Control Command” on page 82 with F_DUPFD cannot be used to duplicate the *scan descriptor*. This is so the system has tight control of the closing of this scan descriptor.
- Data read using this descriptor will be in the *From CCSID* format. If any data is written using this descriptor, it must be in the *From CCSID* format. For more information on CCSIDs see “Coded Character Set Identifier (CCSID) Information” on page 531 in “Integrated File System Scan on Open Exit Program” on page 523.
- It will be a different descriptor than was specified on the close request.
- The oflags for this descriptor are what are passed on this interface.
- It is scoped to the process. However, one can use givedescriptor() and takedescriptor() or sendmsg() and recvmsg() to pass this descriptor to another job or process. Again, that process must complete its use of that descriptor before control is returned to the system from the exit program because the system will close the descriptor when exit program processing is complete. The system will wait on this close attempt until all accesses to this object are closed.
- No other threads in the process, other than those created by the exit program, will be able to access this descriptor.
- It only lives for the life of the exit program invocation. That is, once control is returned from the exit program, it will be destroyed. Therefore, it cannot be stored for later use.

Scan key. The scan key associated with this exit program. The first character of this scan key can not be hex zeros or a blank. For more information on the scan key, see “Scan Key List and Scan Key Signatures” on page 530 in “Integrated File System Scan on Open Exit Program” on page 523.

Scan key signature. The scan key signature associated with the specified *scan key*. For more information on the scan key signature, see “Scan Key List and Scan Key Signatures” on page 530 in “Integrated File System Scan on Open Exit Program” on page 523. If the specified *scan key* already exists in the scan key list, and the exit program is being added to replace an existing exit program entry, then the specified *scan key signature* must match the scan key signature associated with the scan key in the scan key list. If the specified *scan key* already exists in the scan key list, and the exit program is **not** being added to replace an existing exit program entry, then the specified *scan key signature* must match the scan key signature associated with the scan key in the scan key list unless the scan key signature associated with the scan key in the scan key list is all hex zeros. More than one exit program, including exit programs associated with the “Integrated File System Scan on Open Exit Program” on page 523, can have the same scan key signature.

Scan signatures different. Whether the exit program was called because the object’s current scan key signature is different than the appropriate associated signature. When an object is in an independent ASP group, the object scan signature is compared to the associated independent ASP group scan signature. When an object is **not** in an independent ASP group, the object scan signature is compared to the global scan signature. The possible values are:

QP0L_SCAN_NO (x’00’)	The compared signatures are not different.
QP0L_SCAN_YES (x’01’)	The compared signatures are different.

Scan status. The status of the scan processing. This field is only used if the *Update object scan information* field value specifies a value of QP0L_SCAN_YES. The possible values are:

QP0L_SCAN_SUCCESS (x’01’)	The object was scanned and has no failures. If this indicator is returned by all exit programs that were called, the object will be marked as scan successful, and the close operation completes with no errors.

QP0L_SCAN_FAILURE (x'02')	The object was scanned and has at least one failure. If this indicator is returned by at least one of the exit programs that was called, the object will be marked as scan failure, and the close operation fails. Additionally, only the CCSID or binary indication related to this failing request will be kept in the object scan information, and the rest of the historical CCSID or binary information will be cleared. Once an object has been marked as a failure under this condition, it will not be scanned again until the object's scan signature is different than the global scan key signature or independent ASP group scan key signature as appropriate. Therefore, subsequent requests to work with the object will fail with a scan failure indication. Examples of requests which will fail are opening the object, changing the CCSID of the object, copying the object etc..
QP0L_SCAN_FAIL_WANT_ MODIFY (x'03')	The object was scanned and has at least one failure. However, the exit program wanted to modify the file to correct the failure, but could not because it did not have write access. If this indicator is returned by at least one of the exit programs that was called, the object will be marked as scan failure, and the close operation fails. Additionally, only the CCSID or binary indication related to this failing request will be kept in the object scan information, and the rest of the historical CCSID or binary information will be cleared. Once an object has been marked as a failure under this condition, it will not be scanned again until the object's scan signature is different than the global scan key signature or independent ASP group scan key signature as appropriate, or if a subsequent access would allow write access to be given to the exit program. Therefore, subsequent requests to work with the object will fail with a scan failure indication. Examples of requests which will fail are opening the object, changing the CCSID of the object, copying the object etc..
QP0L_SCAN_MODIFY (x'04')	The object was scanned, one or more failures were found, but the object was modified to remove the failures. If this indicator is returned by at least one of the exit programs that was called, then any exit programs which have previously been called will be called one more time so that they can scan the modified object information. This second call is indicated by an *Additional call* field value. If after this additional call, no failures are found, the object will be marked as scan successful, and the close operation completes with no errors.

If a value other than the possible values is specified, the object will be treated as if the program was not called and the object was not scanned. Therefore, the close operation will continue unless the Scan file systems control (QSCANFSCTL) system value has *ERRFAIL specified which will cause the operation to fail. If the Scan file systems control (QSCANFSCTL) system value has *NOFAILCLO specified, the close operation will not return any failure indication. If *NOFAILCLO is not specified, the close operation will fail with error code [ESCANFAILURE].

To CCSID. The CCSID value that was specified on the open request associated with this close request. Therefore, this will be the CCSID in which data was returned (when the user was reading from the object), or the CCSID in which data was be supplied (when the user was writing to the object). Therefore, the exit program should be converting the data to this CCSID since this is how the data was presented to the user after their open request completed. For more information on CCSIDs and conversions, see "Coded Character Set Identifier (CCSID) Information" on page 531 in "Integrated File System Scan on Open Exit Program" on page 523.

Note: If the *To CCSID* and *From CCSID* values match, it is the same as if the object was opened in binary.

Update object scan information. Whether the scan information associated with the object should be updated or not. The object scan information includes the following:

- Scan status for the object.
- Scan signature associated with the object scan status.
- The *To CCSID* value of the object which was scanned or if the object was scanned in binary.

Note: Actually, the last two To CCSID values which have been scanned will be maintained as well as a separate indication of binary scans.

The possible values are:

QP0L_SCAN_NO (x'00')	The object scan information should not be updated. This might be used when the object was not actually scanned by the exit program, perhaps because it did not need to be, or perhaps because a deferred scan was initiated.
QP0L_SCAN_YES (x'01')	The object scan information should be updated. When this value is set, then the values in the *Scan status* field and *Failing CCSID* are used. If at least one exit program specified this value, then the object scan information will be updated.

If a value other than the possible values is specified, a value of QP0L_SCAN_NO is assumed.

User profile. The exit program will be called under this user profile. Therefore, this user profile should have *USE authority to the exit program, and *EXECUTE authority to the exit program library. If the user profile is not valid or accessible at the time the exit program is called, the object will be treated as if the program was not called and the object was not scanned. Therefore, the close operation will continue unless the Scan file systems control (QSCANFSCTL) system value has *ERRFAIL specified which will cause the operation to fail. If the Scan file systems control (QSCANFSCTL) system value has *NOFAILCLO specified, the close operation will not return any failure indication. If *NOFAILCLO is not specified, the close operation will fail with error code [ESCANFAILURE]. The first character of the user profile can not be hex zeros or a blank.

Note: The system will not do any additional verification that this specified profile has authority to the object for which the exit program is being called when that exit program is being called, even when the access levels for the object are upgraded to include write. By registering this exit program, you are indicating this is acceptable.

Usage Notes

1. When the exit program is executing (including any created threads), if it does any operations on other objects which might normally trigger another call to a scan-related exit program, the scan-related exit program will **not** be called, and it will be treated as if no scanning occurred for the object. For example, if the exit program opens a separate object, that object will not be scanned as part of that open request, even if an exit program is registered to the QIBM_QP0L_SCAN_OPEN exit point. If however, that object has previously failed a scan, then the operation will fail with error code [ESCANFAILURE].
2. When the exit program is executing (including any created threads), if it does any opens of other objects, then the descriptors which will be returned will come from the same table of descriptors that the *Scan descriptor* is derived from. Therefore, customer application code will not be impacted by 'regular' descriptors being used and possibly reaching an application specified limit on the number of descriptors which can be used. Additionally, the exit program will not be able to use any of the 'regular' descriptors when it or any of its created threads are executing. That is, it will not be able to access any objects which have been opened outside the scope of the exit program execution. Any attempts to do so will fail with error code [EBADF].
3. When the following APIs are called from the thread executing the exit program and any of its created threads, the table of *Scan descriptors*, will not be inherited by the spawned process.
   - spawn()—Spawn Process
   - spawnp()—Spawn Process with Path

   Therefore, when the following APIs are called from the thread executing the exit program and any of its created threads, the descriptors returned by these APIs will only work within the same process.
   - "pipe()—Create an Interprocess Channel" on page 221—Create Interprocess Channel
   - "Qp0zPipe()—Create Interprocess Channel with Sockets" on page 419—Create Interprocess Channel with Sockets()

4. When the exit programs are executing (including any created threads), signals are blocked from being delivered to a thread. When a signal is blocked, the signal-handling action associated with the signal is not taken. The signal remains pending until all exit programs have completed execution. For more information, see Signal concepts.
5. When the following APIs are called from the thread executing the exit program and any of its created threads, they will fail with the listed error code.
   - “DosSetFileLocks()—Lock and Unlock a Byte Range of an Open File” on page 47— [ENOTSUP]
   - “DosSetFileLocks64()—Lock and Unlock a Byte Range of an Open File (Large File Enabled)” on page 51— [ENOTSUP]
   - “DosSetRelMaxFH()—Change Maximum Number of File Descriptors” on page 53 — [ERROR_GEN_FAILURE]
   - “dup2()—Duplicate Open File Descriptor to Another Descriptor” on page 58 — [ENOTSUP]
   - “fcntl()—Perform File Control Command” on page 82 with F_SETLK, F_SETLK64, F_SETLKW or F_SETLKW64 — [ENOTSUP]
   - » recvmsg()—Receive Data or Descriptors or Both— [ENOTSUP]
   - socketpair()—Create a Pair of Sockets— [ENOTSUP] «
   - takedescriptor()—Receive Socket Access from Another Job — [ENOTSUP]
6. Unpredictable results will occur if the select()—Wait for events on multiple sockets » or poll()—Wait for events on multiple descriptors « APIs and any of their associated type and macro definitions are used in the thread executing the exit program and any of its created threads. Therefore, these interfaces should not be used under these conditions.
7. It is recommended that the exit program use the large-file enabled APIs such as “lseek64()—Set File Read/Write Offset (Large File Enabled)” on page 161 to work with the *scan descriptor* as these APIs will work with any size object.
8. If Kerberos is configured on the system, then the thread executing the exit program and any of its created threads will not be able to access objects in any file systems which use Kerberos for authentication. If they do, the operation will fail with error code [ENOTSUP]. E.g. the exit program cannot access objects in the QFileSvr.400 file system when Kerberos is configured.
9. The exit program should not call the open or close API interfaces on the object represented by the *scan descriptor*. If this is done from the thread executing the exit program, then [EDEADLK] will be returned. If the object is opened or closed from any other process or thread, that process or thread will wait until this invocation’s scan is completed.

Related Information

- The <**qp0lscan.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- Change Scan Signature (QP0LCHSG) API
- “Integrated File System Scan on Open Exit Program” on page 523
- “Qp0lGetAttr()—Get Attributes” on page 326
- “Qp0lSetAttr()—Set Attributes” on page 403
- Retrieve Scan Signature (QP0LRTSG) API
- Retrieve System Values (QWCRSVAL) API

Exit program introduced: V5R3

Top | “Integrated File System APIs,” on page 1 | APIs by category

Integrated File System Scan on Open Exit Program

Required Parameter Group:

1	Integrated file system open exit information	Input	Char(*)
2	Status information	Output	Char(*)

QSYSINC Member Name: QP0LSCAN
Exit Point Name: QIBM_QP0L_SCAN_OPEN
Exit Point Format Name: SCOP0100

The integrated file system scan on open exit program is called to do scan processing when an integrated file system object is opened under the following conditions.

The exit program will **not** be called if:

- No exit programs exist for this exit point.
- -or- the Scan file systems (QSCANFS) system value has *NONE specified so that no file systems will be scanned.
- -or- the object was marked to not be scanned and a scan is not required because the object was restored. (See **Note** 2)
- -or- the object is being opened for write access only.
- -or- the object is being truncated as part of the open request.
- -or- the object is being used as a network server storage space or as a virtual volume. From the perspective of the server, these objects appear as byte stream files within the integrated file system.
- -or- the object is not being accessed from a file server, and the Scan file systems control (QSCANFSCTL) system value has *FSVRONLY specified so that only file server accesses are scanned. (See **Note** 2)
- -or- the object is in a file system that has not completely converted to the *TYPE2 directory format. For information on the *TYPE2 directory format, see the Convert Directory (CVTDIR) command and the Integrated file system information in the Files and file systems topic.

If the previous conditions have been met, the exit program will be called if:

- The object has never been scanned.
- -or- the object's data has been modified since the last time it was scanned. Data modifications include writes, memory map writes, truncates or clears.
- -or- the CCSID of the object has been modified since the last time it was scanned.
- -or- the To CCSID specified on the open request is different than the last two To CCSIDs that were specified and previously scanned for this object.
- -or- the object is being opened in binary, and it has not previously been scanned in binary.
- -or- there have been updates to the scanning software and the object was not marked to be scanned only if the object changed (See **Note** 2). Updates to scanning software occur by either registering additional exit programs for the scan-related exit points, or by calling Change Scan Signature (QP0LCHSG) API to update the scan key signature associated with existing exit program scan keys.

For more information on open processing, as well as CCSID values, see "open()—Open File" on page 195. For more information on the scan-related attributes which can be set for objects, see "Qp0lSetAttr()—Set Attributes" on page 403 For more information on the integrated file system scan processing and various options, see the Integrated file system information in the Files and file systems topic

The exit point supports a maximum of 50 exit programs. For information about adding an exit program to an exit point, see the Registration Facility.

Notes:

1. If the integrated file system exit program returns any error messages or if any errors are received when attempting to call the exit program, the object will be treated as if the program was not called and the object was not scanned. Therefore, the open operation will continue unless the QSCANFSCTL system value has *ERRFAIL specified which will cause the operation to fail.
2. If the oflags specified when the object was opened include the O_FORCE_SCAN value, then one or more of the following conditions will be ignored when determining whether the integrated file system scan-related exit programs will be called:
   - The QSCANFSCTL system value specification of *FSVRONLY.
   - The object was marked to not be scanned (e.g. scan attribute of *NO).
   - The object was marked to be scanned only if the object changed (e.g. scan attribute of *CHGONLY).

   For example, an object is opened that has a scan attribute of *YES, and the open request is not through the file servers when *FSVRONLY is specified. If O_FORCE_SCAN is specified on that open request, the object will be scanned if all the remaining conditions are met. Similarly, if an object that has a scan attribute of *NO or *CHGONLY is opened with O_FORCE_SCAN specified, the object will be scanned if all the remaining conditions are met. See "Using the oflag Parameter" on page 197 in "open()—Open File" on page 195 for more information on oflags.

Restrictions

- Only objects of type *STMF that are in the "root" (/), QOpenSys, and user-defined file systems that have completely converted to the *TYPE2 directory format are scanned. For information on *TYPE2 directories, see the Convert Directory(CVTDIR) command and the Integrated file system information in the Files and file systems topic.
- The exit programs will not be called during an IPL or the vary-on of an independent Auxiliary Storage Pool (ASP).
- During the call to the exit programs, the ASP group associated with the thread will not be able to be changed.
- The exit programs must exist in the system ASP or in a basic user ASP. They cannot exist in an independent ASP. Any ASP group could be associated with the thread when the exit program is called. If the exit program is not found, the object will be treated as if the program was not called and the object was not scanned. Therefore, the open operation will continue unless the Scan file systems control (QSCANFSCTL) system value has *ERRFAIL specified which will cause the operation to fail.
- The exit programs could be called from an exit point within a multi-threaded job and must be written to be threadsafe.

Authorities and Locks

User Profile Authority

*ALLOBJ (all object) and *SECADM (security administrator) special authorities to add exit programs to the registration facility

*ALLOBJ and *SECADM special authorities to remove exit programs from the registration facility

Program Data

When you register the exit program, the following program data must be provided. The following table shows the structure of the program data information. For a description of the fields in this format, see "Field Descriptions" on page 526. This structure is defined in header file qp0lscan.h as data type Qp0l_Scan_Program_Data_t.

Offset			
Dec	**Hex**	**Type**	**Field**
0	0	Char(10)	User profile
10	A	Char(20)	Scan key

Offset		Type	Field
Dec	Hex		
30	1E	Char(12)	Scan key signature

Required Parameter Group

Integrated file system open exit information
INPUT; CHAR(*)

Information that is needed by the exit program to do its object scan processing. For details, see "Format of Integrated File System Open Exit Information (Input)."

Status information
OUTPUT; CHAR(*)

Information that is returned by the exit program indicating what scan processing has occurred. For details, see "Format of Status Information (Output)."

Format of Integrated File System Open Exit Information (Input)

The following table shows the structure of the integrated file system open exit information for exit point format SCOP0100. For a description of the fields in this format, see "Field Descriptions" on page 526. This structure is defined in header file qp0lscan.h as data type Qp0l_Scan_Exit_Information_t.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Integrated file system open exit information length
4	4	CHAR(20)	Exit point name
24	18	CHAR(8)	Exit point format name
32	20	BINARY(4)	Length of status information
36	24	BINARY(4)	Scan descriptor
40	28	BINARY(4), UNSIGNED	From CCSID
44	2C	BINARY(4), UNSIGNED	To CCSID
48	30	BINARY(4), UNSIGNED	Last failure CCSID
52	34	BINARY(4)	Oflags
56	38	CHAR(16)	File ID
72	48	CHAR(10)	Object type
82	52	CHAR(1)	File system
83	53	CHAR(1)	Additional call
84	54	CHAR(1)	Object modified since last scan
85	55	CHAR(1)	Scan signatures different
86	56	CHAR(1)	Call after previous failure

Format of Status Information (Output)

The following table shows the structure of the status information. For a description of the fields in this format, see "Field Descriptions" on page 526. This structure is defined in header file qp0lscan.h as data type Qp0l_Scan_Status_Information_t.

Offset		Type	Field
Dec	Hex		
0	0	BINARY(4)	Open scan status information length
4	4	BINARY(4), UNSIGNED	Failing CCSID
8	8	CHAR(1)	Update object scan information
9	9	CHAR(1)	Scan status

Field Descriptions

Additional call. Whether the exit program was called an additional time because another "Integrated File System Scan on Open Exit Program" on page 523 that was called has indicated the object was modified. See the *scan status* field for this modify indication. The possible values are:

QP0L_SCAN_CALL_FIRST (x'00') The first call to the exit program.

QP0L_SCAN_CALL_ADDL (x'01') An additional call to the exit program because another exit program has indicated the object was modified.

Call after previous failure. Whether the exit program was called after the object had previously been scanned and a failure detected. The possible values are:

QP0L_SCAN_NO (x'00') This is not a call after a previous scan failure.

QP0L_SCAN_YES (x'01') This is a call after a previous scan failure. The *Last failure CCSID* field in conjunction with the *From CCSID* indicate the CCSID or binary indication of the failing scan request.

Note: If the *Failing CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

Exit point format name. The format name for the integrated file system scan on open exit program. The possible format name follows:

SCOP0100 The format name that is used while an object is being opened.

Exit point name. The name of the exit point that is calling the exit program.

Failing CCSID. This field only has meaning if the *Call after previous failure* field had a value of QP0L_SCAN_YES when the exit program was called, and if the *Update object scan information* field has a value of QP0L_SCAN_YES, and if the *Scan status* field has a value of QP0L_SCAN_FAILURE or QP0L_SCAN_FAIL_WANT_MODIFY. When the *Call after previous failure* had a value of QP0L_SCAN_YES, then the scan exit program should verify that the object does not have any problems when scanned using both the *To CCSID* and *Last failure CCSID* values. If either scan fails, then this field should be filled in with the failing CCSID which will be stored as part of the object scan information with the failure indication. If the value of this field does not match either of the two input CCSID fields, then the *To CCSID* value will be used. If more than one exit program indicates a failure, the failing CCSID value which will be preserved is from the last exit program which scanned the object and indicated a failure. For more information on CCSIDs and conversions, see "Coded Character Set Identifier (CCSID) Information" on page 531.

Note: If the *Failing CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

File ID. A unique identifier associated with the object that is being opened. A file ID can be used with "Qp0lGetPathFromFileID()—Get Path Name of Object from Its File ID" on page 351, to retrieve an object's path name.

File system. The file system that the object being scanned is in. The possible value follows:

QP0L_SCAN_ROOT_QOPENSYS_UDFS	The object is in the "root" (/), QOpenSys, or a user-defined file system.

From CCSID. The CCSID value that the data is in on the system itself. Therefore, this will be the CCSID in which data is to be returned (when reading from the object using the *Scan descriptor*), or the CCSID in which data is being supplied (when writing to the object using the *Scan descriptor*). For more information on CCSIDs and conversions, see "Coded Character Set Identifier (CCSID) Information" on page 531.

Integrated file system open exit information length. The length in bytes of all data passed to the integrated file system open exit program.

Last failure CCSID. The CCSID value that was specified when this object was last scanned and indicated a scan failure. This field only has meaning if the *Call after previous failure* field has a value of QP0L_SCAN_YES. Therefore, this would have been the CCSID in which data was to have been returned (when the user was to be reading from the object), or the CCSID in which data was to have been supplied (when the user was to be writing to the object). However, that request failed for this CCSID. This is now being returned so that this CCSID can also be scanned, if it is different than the *To CCSID* value. For more information on CCSIDs and conversions, see "Coded Character Set Identifier (CCSID) Information" on page 531.

Note: If the *Last failure CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary.

Length of status information. The length in bytes allocated for the returned status information.

Object modified since last scan. Whether the exit program was called because the objects data or CCSID has been modified since it was last scanned. Examples of object data or CCSID modifications are: writing to the object, directly or through memory mapping; truncating the object; clearing the object; and changing the objects CCSID attribute, etc.. The possible values are:

QP0L_SCAN_NO (x'00')	The object has not been modified since it was last scanned.
QP0L_SCAN_YES (x'01')	The object has been modified since it was last scanned.

Object type. The object type. See Control Language (CL) information in the iSeries Information Center for descriptions of all object types.

Oflags. The oflags that were specified on the open request with the following exceptions. For a description of all possible oflag values, see "open()—Open File" on page 195

- If the oflags do not contain write access, the system will attempt to upgrade the access intent to include write, unless the Scan file systems control (QSCANFSCTL) system value has *NOWRTUPG specified or the object is not eligible for write access. If the upgrade is not attempted or is unsuccessful, the access intent matches the users invocation. If it is successful, the write access intent is included in this oflag information. This upgrade would be useful if the exit program wanted to modify the object to correct any problems found while scanning.
- The CCSID related flags will have been removed. This includes O_TEXTDATA, O_CCSID, O_CODEPAGE, and O_TEXT_CREATE.
- The synchronization flags will have been removed. This includes O_SYNC, O_DSYNC, and O_RSYNC.

Open scan status information length. The length in bytes of all data returned from the integrated file system open exit program. The only valid value for this field is 10. If anything else is specified, the object will be treated as if the program was not called and the object was not scanned. Therefore, the open operation will continue unless the Scan file systems control (QSCANFSCTL) system value has *ERRFAIL specified which will cause the operation to fail.

Scan descriptor. A descriptor representing the object that is being opened. This scan descriptor has the following characteristics:

- It can be used to do any read processing on the object being processed. Reads using this descriptor will not update the last access timestamp information for the object.
- It can be used to do any write processing on the object being processed. If write processing is done by the exit program, the exit program should indicate QP0L_SCAN_MODIFY in the *Scan status* field. If it does not, the object's scan information will be cleared as if the objects data has been modified.
- It cannot be used to memory map the object, see "mmap()—Memory Map a File" on page 179
- It cannot be used to close the object using "close()—Close File or Socket Descriptor" on page 34. When control returns from the exit program, the system code will do the close of this *scan descriptor*. The system will wait on this close attempt until all accesses to this object are closed. Therefore, if the exit program uses givedescriptor()—Pass Descriptor Access to Another Job and takedescriptor()—Receive Socket Access from Another Job or sendmsg()—Send Data or Descriptors or Both and recvmsg()—Receive Data or Descriptors or Both to pass the descriptor to another job, the job which used takedescriptor() or recvmsg() must close that descriptor when it is done processing, else the system will be waiting for that close.
- "dup()—Duplicate Open File Descriptor" on page 55 and "fcntl()—Perform File Control Command" on page 82 with F_DUPFD cannot be used to duplicate the *scan descriptor*. This is so the system has tight control of the closing of this scan descriptor.
- Data read using this descriptor will be in the *From CCSID* format. If any data is written using this descriptor, it must be in the *From CCSID* format. For more information on CCSIDs see "Coded Character Set Identifier (CCSID) Information" on page 531.
- It will be a different descriptor than will actually be returned to the user, if the open is ultimately successful.
- The oflags for this descriptor are what are passed on this interface.
- It is scoped to the process. However, one can use givedescriptor() and takedescriptor() or sendmsg() and recvmsg() to pass this descriptor to another job or process. Again, that process must complete its use of that descriptor before control is returned to the system from the exit program because the system will close the descriptor when exit program processing is complete. The system will wait on this close attempt until all accesses to this object are closed.
- No other threads in the process, other than those created by the exit program, will be able to access this descriptor.
- It only lives for the life of the exit program invocation. That is, once control is returned from the exit program, it will be destroyed. Therefore, it cannot be stored for later use.

Scan key. The scan key associated with this exit program. The first character of this scan key can not be hex zeros or a blank. For more information on the scan key, see "Scan Key List and Scan Key Signatures" on page 530

Scan key signature. The scan key signature associated with the specified *scan key*. For more information on the scan key signature, see "Scan Key List and Scan Key Signatures" on page 530. If the specified *scan key* already exists in the scan key list, and the exit program is being added to replace an existing exit program entry, then the specified *scan key signature* must match the scan key signature associated with the scan key in the scan key list. If the specified *scan key* already exists in the scan key list, and the exit program is **not** being added to replace an existing exit program entry, then the specified *scan key signature* must match the scan key signature associated with the scan key in the scan key list unless the scan key

signature associated with the scan key in the scan key list is all hex zeros. More than one exit program, including exit programs associated with the "Integrated File System Scan on Close Exit Program" on page 513, can have the same scan key signature.

Scan signatures different. Whether the exit program was called because the object's current scan key signature is different than the appropriate associated signature. When an object is in an independent ASP group, the object scan signature is compared to the associated independent ASP group scan signature. When an object is **not** in an independent ASP group, the object scan signature is compared to the global scan signature. The possible values are:

QP0L_SCAN_NO (x'00')	The compared signatures are not different.
QP0L_SCAN_YES (x'01')	The compared signatures are different.

Scan status. The status of the scan processing. This field is only used if the *Update object scan information* field value specifies a value of QP0L_SCAN_YES. The possible values are:

QP0L_SCAN_SUCCESS (x'01')	The object was scanned and has no failures. If this indicator is returned by all exit programs that were called, the object will be marked as scan successful, and the open operation completes with no errors.
QP0L_SCAN_FAILURE (x'02')	The object was scanned and has at least one failure. If this indicator is returned by at least one of the exit programs that was called, the object will be marked as scan failure, and the open operation fails. Additionally, only the CCSID or binary indication related to this failing request will be kept in the object scan information, and the rest of the historical CCSID or binary information will be cleared. Once an object has been marked as a failure under this condition, it will not be scanned again until the object's scan signature is different than the global scan key signature or independent ASP group scan key signature as appropriate. Therefore, subsequent requests to work with the object will fail with a scan failure indication. Examples of requests which will fail are opening the object, changing the CCSID of the object, copying the object etc..
QP0L_SCAN_FAIL_WANT_ MODIFY (x'03')	The object was scanned and has at least one failure. However, the exit program wanted to modify the file to correct the failure, but could not because it did not have write access. If this indicator is returned by at least one of the exit programs that was called, the object will be marked as scan failure, and the open operation fails. Additionally, only the CCSID or binary indication related to this failing request will be kept in the object scan information, and the rest of the historical CCSID or binary information will be cleared. Once an object has been marked as a failure under this condition, it will not be scanned again until the object's scan signature is different than the global scan key signature or independent ASP group scan key signature as appropriate, or if a subsequent access would allow write access to be given to the exit program. Therefore, subsequent requests to work with the object will fail with a scan failure indication. Examples of requests which will fail are opening the object, changing the CCSID of the object, copying the object etc..
QP0L_SCAN_MODIFY (x'04')	The object was scanned, one or more failures were found, but the object was modified to remove the failures. If this indicator is returned by at least one of the exit programs that was called, then any exit programs which have previously been called will be called one more time so that they can scan the modified object information. This second call is indicated by an *Additional call* field value. If after this additional call, no failures are found, the object will be marked as scan successful, and the open operation completes with no errors.

If a value other than the possible values is specified, the object will be treated as if the program was not called and the object was not scanned. Therefore, the open operation will continue unless the Scan file systems control (QSCANFSCTL) system value has *ERRFAIL specified which will cause the operation to fail.

To CCSID. The CCSID value that was specified on the open request. Therefore, this will be the CCSID in which data will be returned (when the user will be reading from the object), or the CCSID in which data will be supplied (when the user will be writing to the object). Therefore, the exit program should be converting the data to this CCSID since this is how the data will be presented to the user if the open request completes successfully. For more information on CCSIDs and conversions, see "Coded Character Set Identifier (CCSID) Information" on page 531.

Note: If the *To CCSID* and *From CCSID* values match, it is the same as if the object will be opened in binary.

Update object scan information. Whether the scan information associated with the object should be updated or not. The object scan information includes the following:

- Scan status for the object.
- Scan signature associated with the object scan status.
- The *To CCSID* value of the object which was scanned or if the object was scanned in binary.

  Note: Actually, the last two To CCSID values which have been scanned will be maintained as well as a separate indication of binary scans.

The possible values are:

QP0L_SCAN_NO (x'00')	The object scan information should not be updated. This might be used when the object was not actually scanned by the exit program, perhaps because it did not need to be, or perhaps because a deferred scan was initiated.
QP0L_SCAN_YES (x'01')	The object scan information should be updated. When this value is set, then the values in the *Scan status* field and *Failing CCSID* are used. If at least one exit program specified this value, then the object scan information will be updated.

If a value other than the possible values is specified, a value of QP0L_SCAN_NO is assumed.

User profile. The exit program will be called under this user profile. Therefore, this user profile should have *USE authority to the exit program, and *EXECUTE authority to the exit program library. If the user profile is not valid or accessible at the time the exit program is called, the object will be treated as if the program was not called and the object was not scanned. Therefore, the open operation will continue unless the Scan file systems control (QSCANFSCTL) system value has *ERRFAIL specified which will cause the operation to fail. The first character of the user profile can not be hex zeros or a blank.

Note: The system will not do any additional verification that this specified profile has authority to the object for which the exit program is being called when that exit program is being called, even when the access levels for the object are upgraded to include write. By registering this exit program, you are indicating this is acceptable.

Scan Key List and Scan Key Signatures

A list of *scan keys* and associated *scan key signatures* will be used to help minimize unnecessary scan calls, while allowing users to ensure scans occur when needed. The *scan key* list and *scan key signature* will allow an association of scanning software level with the various scan-related exit programs ("Integrated File System Scan on Close Exit Program" on page 513 and "Integrated File System Scan on Open Exit Program" on page 523). Updates to this information will allow the system to increment its global scan signature field to reflect the software updates.

The system will maintain a global scan signature field and independent ASP group scan signature fields. Each integrated file system object which supports scanning will have an object scan signature field.

The global scan signature indicates the state or level of the scanning software. It will or will not be modified under the following rules:

- When the scan-related exit programs are added or registered, the user specifies a scan key and a scan key signature. These values are added to the scan key list. If the scan key has previously been specified, e.g. for a different exit program registration, then the global scan signature will only be incremented if the specified scan key signature is not hex zero. If the scan key has not previously been specified, and the scan key signature is not a hex zero value, the global scan signature will be incremented.
- By calling the Change Scan Signature (QP0LCHSG) API to specify that a new scan key signature be associated with a specific scan key. This will cause the system to update the scan key list and increment the current global scan signature value.
- When the scan-related exit programs are removed, the user specifies a scan key and a scan key signature. These values are removed from the scan key list if no other scan-related exit programs are registered that have that scan key. Removing entries from the scan key list does **not** update the global scan signature.

The independent ASP group scan signature indicates the state of the scanning software as well. Since it moves with the independent ASP group, it represents the state of the scanning code software in relationship to when and where that independent ASP group was varied on. The independent ASP group scan key list and independent ASP group scan signature will or will not be modified under the following rules:

- If the independent ASP group is available and online, the independent ASP group scan key list will be updated whenever the system scan key list is updated. Any changes to the independent ASP group scan key list will cause the independent ASP group scan signature to be incremented under the same rules as to when the global scan signature is updated.
- If the independent ASP group is varied on after any global scan key list changes, then when the first scannable integrated file system object on the independent ASP group is opened or its scan information is retrieved, the independent ASP group scan key list will be compared to the global scan key list.
  - If the global scan key list has more scan keys or different scan key signatures than the independent ASP group scan key list has, then the independent ASP group scan list will be updated to match. Additionally, the independent ASP group scan signature will be incremented.
  - If the global scan key list is a proper subset of the scan keys and scan key signatures in the independent ASP group scan key list, then the independent ASP group scan list will be updated to match. However, the independent ASP group scan signature will not be incremented. If the global scan key list exactly matchs the scan keys and scan key signatures in the independent ASP group scan key list, then no changes are made.

It is highly recommended that the scanning software level of support which is indicated by scan keys and scan key signatures be maintained the same across all systems in the independent ASP Cluster group. See Cluster for more information.

When an object in an independent ASP group is about to be scanned or its scan information is retrieved, the object scan signature will be compared to the associated independent ASP group scan signature. When an object is **not** in an independent ASP group, the object scan signature will be compared to the global scan signature value.

When an object is successfully scanned, the object scan signature will be updated to match the global scan signature or independent ASP group scan signature when scanning was begun as appropriate. Other associated fields will be updated as well as described in *Update object scan information*.

Coded Character Set Identifier (CCSID) Information

The CCSID values presented on this interface have the following meanings and inter-relationships. The *From CCSID* represents the value for the data that is stored in the object. Therefore, when discussing reading and writing in the *From CCSID* format, it means the data is read or written as is, no conversion occurs between what is given to or returned by the system, and the data in the object itself. The *scan descriptor* that is passed to the exit program is not an open instance which provides CCSID conversion.

But, when the object is ultimately opened, the file descriptor that is returned will include conversion using the value in *To CCSID*. If the *To CCSID* and *From CCSID* values match, it is the same as if the object would have been opened in binary. If the object is not being opened in binary, the scan exit program should do its scanning using the *To CCSID* value, and can use the appropriate APIs to do the conversion. If the scan succeeds or fails, then the CCSID which is preserved with the scan status information is the To CCSID, except for the following case. If the *Call after previous failure* field has a value of QP0L_SCAN_YES, and the value in the *Last Failure CCSID* is different than *To CCSID*, then the scan exit program should also scan the object data using the *Last Failure CCSID*. In this case, if the scan succeeds, then the CCSID which is preserved with the scan status information is the *To CCSID*. If the scan fails, then the CCSID which is preserved with the scan status information is the *Failing CCSID*.

For more information on CCSIDs and conversions, see "open()—Open File" on page 195 and Globalization topic.

Usage Notes

1. When the exit program is executing (including any created threads), if it does any operations on other objects which might normally trigger another call to a scan-related exit program, the scan-related exit program will **not** be called, and it will be treated as if no scanning occurred for the object. For example, if the exit program opens a separate object, that object will not be scanned as part of that open request, even if an exit program is registered to the QIBM_QP0L_SCAN_OPEN exit point. If however, that object has previously failed a scan, then the operation will fail with error code [ESCANFAILURE].
2. When the exit program is executing (including any created threads), if it does any opens of other objects, then the descriptors which will be returned will come from the same table of descriptors that the *Scan descriptor* is derived from. Therefore, customer application code will not be impacted by 'regular' descriptors being used and possibly reaching an application specified limit on the number of descriptors which can be used.

   Additionally, the exit program will not be able to use any of the 'regular' descriptors when it or any of its created threads are executing. That is, it will not be able to access any objects which have been opened outside the scope of the exit program execution. Any attempts to do so will fail with error code [EBADF].
3. When the following APIs are called from the thread executing the exit program and any of its created threads, the table of *Scan descriptors*, will not be inherited by the spawned process.
   - spawn()—Spawn Process
   - spawnp()—Spawn Process with Path

   Therefore, when the following APIs are called from the thread executing the exit program and any of its created threads, the descriptors returned by these APIs will only work within the same process.
   - "pipe()—Create an Interprocess Channel" on page 221—Create Interprocess Channel
   - "Qp0zPipe()—Create Interprocess Channel with Sockets" on page 419—Create Interprocess Channel with Sockets()
4. When the exit programs are executing (including any created threads), signals are blocked from being delivered to a thread. When a signal is blocked, the signal-handling action associated with the signal is not taken. The signal remains pending until all exit programs have completed execution. For more information, see Signal concepts.
5. When the following APIs are called from the thread executing the exit program and any of its created threads, they will fail with the listed error code.
   - "DosSetFileLocks()—Lock and Unlock a Byte Range of an Open File" on page 47— [ENOTSUP]
   - "DosSetFileLocks64()—Lock and Unlock a Byte Range of an Open File (Large File Enabled)" on page 51— [ENOTSUP]
   - "DosSetRelMaxFH()—Change Maximum Number of File Descriptors" on page 53 — [ERROR_GEN_FAILURE]
   - "dup2()—Duplicate Open File Descriptor to Another Descriptor" on page 58 — [ENOTSUP]

- “fcntl()—Perform File Control Command” on page 82 with F_SETLK, F_SETLK64, F_SETLKW or F_SETLKW64 — [ENOTSUP]
- recvmsg()—Receive Data or Descriptors or Both— [ENOTSUP]
- socketpair()—Create a Pair of Sockets— [ENOTSUP]
- takedescriptor()—Receive Socket Access from Another Job — [ENOTSUP]

6. Unpredictable results will occur if the select()—Wait for events on multiple sockets or poll()—Wait for events on multiple descriptors APIs and any of their associated type and macro definitions are used in the thread executing the exit program and any of its created threads. Therefore, these interfaces should not be used under these conditions.
7. It is recommended that the exit program use the large-file enabled APIs such as “lseek64()—Set File Read/Write Offset (Large File Enabled)” on page 161 to work with the *scan descriptor* as these APIs will work with any size object.
8. If Kerberos is configured on the system, then the thread executing the exit program and any of its created threads will not be able to access objects in any file systems which use Kerberos for authentication. If they do, the operation will fail with error code [ENOTSUP]. E.g. the exit program cannot access objects in the QFileSvr.400 file system when Kerberos is configured.
9. The exit program should not call the open or close API interfaces on the object represented by the *scan descriptor*. If this is done from the thread executing the exit program, then [EDEADLK] will be returned. If the object is opened or closed from any other process or thread, that process or thread will wait until this invocation’s scan is completed.

Related Information

- The <**qp0lscan.h**> file (see “Header Files for UNIX-Type Functions” on page 537)
- Change Scan Signature (QP0LCHSG) API
- “Integrated File System Scan on Close Exit Program” on page 513
- “Qp0lGetAttr()—Get Attributes” on page 326
- “Qp0lSetAttr()—Set Attributes” on page 403
- Retrieve Scan Signature (QP0LRTSG) API
- Retrieve System Values (QWCRSVAL) API

Exit program introduced: V5R3

Top | “Integrated File System APIs,” on page 1 | APIs by category

Process a Path Name Exit Program

Required Parameter Group:

1	Selection status pointer	Input	BINARY(4)
2	Error value pointer	Input	BINARY(4)
3	Return value pointer	Output	BINARY(4)
4	Object name pointer	Input	CHAR(*)
5	Function control block pointer	Input	CHAR(*)

The Process a Path Name exit program is a user-specified exit program that is called by the **Qp0lProcessSubtree()** function for each object in the API’s search that meets the caller’s selection criteria. This exit program can be either a procedure or program.

When the user exit program is given control, it can call other APIs, build lists or tables, or do other processing. Since the API passes the names of all the children objects to the user exit program before passing the name of the parent, the user exit program can also delete directories.

If the exit program encounters an error during processing, it returns a valid *errno* in the Return value pointer field, that **Qp0lProcessSubtree()** returns to its caller. When its processing is complete, the exit program return code is set to tell **Qp0lProcessSubtree()** to do one of the following:

- End processing.
- Continue processing by calling the exit program again with the next object from the same directory.
- Continue processing by calling the exit program again, but not with objects from the same directory. In this case, **Qp0lProcessSubtree()** moves to the next directory or object that meets the specified criteria and calls the exit program with it.

If **Qp0lProcessSubtree()** encounters any problems in resolving to a user exit program, **Qp0lProcessSubtree()** ends and returns to its caller. If **Qp0lProcessSubtree()** encounters any errors with any other parameters, it ends and returns control to its caller, after a call to the user exit program. This call allows the exit program to perform any desired cleanup before **Qp0lProcessSubtree()** ends. Use the *Err_recovery_action* parameter of **Qp0lProcessSubtree()** to set other conditions for calling or not calling the user exit program.

Storage referred to by the Selection status pointer, Error value pointer, Return value pointer, or the Object name pointer when the Process a Path Name exit program is called, are destroyed or reused when **Qp0lProcessSubtree()** regains control.

See “Qp0lProcessSubtree()—Process a Path Name” on page 356 for more information.

Authorities and Locks

None.

Parameters

Selection status pointer
INPUT; BINARY(4)

A pointer to an unsigned integer. This pointer indicates whether **Qp0lProcessSubtree()** encountered any problems in processing. Valid values follow:

0 ***QP0L_SELECT_OK:*** Indicates that no problems were encountered during the selection of the current object.

1 ***QP0L_SELECT_DONE:*** Indicates that the last object was processed and that this is the last call to the Process a Path Name exit program. The Object name pointer parameter is set to NULL.

2 ***QP0L_SELECT_NOT_OK:*** Indicates that **Qp0lProcessSubtree()** has encountered an error but that the Process a Path Name exit program can decide if the operation should continue or end. The Error value pointer parameter points to a valid *errno*.

3 ***QP0L_SELECT_FAILED:*** Indicates that **Qp0lProcessSubtree()** has encountered an unrecoverable error and that **Qp0lProcessSubtree()** will return to its caller when it regains control. The Error value pointer parameter points to a valid *errno*.

Error value pointer
INPUT; BINARY(4)

A pointer to a valid *errno* that describes any problems encountered by the **Qp0lProcessSubtree()** API during the processing of the current object. If no problems were encountered, the error value is zero. Any valid *errno* can be passed in this field.

Return value pointer
OUTPUT; BINARY(4)

A pointer to a value from the Process a Path Name exit program that instructs the API to continue or to end processing. Valid values follow.

0 **Process a Path Name exit program** was successful.

-1	**Process a Path Name exit program** was successful. **Qp0lProcessSubtree()** should skip processing any remaining objects in this directory and move on to process objects in other directories.
> 0 (an errno)	**Process a Path Name exit program** was not successful. **Qp0lProcessSubtree()** ends.

Object name pointer
INPUT; CHAR(*)

A pointer to the path name structure that contains the fully qualified name of the object being processed by **Qp0lProcessSubtree()**. For more information on this structure, see Path Name Format. The Path_Type flag defined in the qlg.h header file must be used to determine whether the Object name pointer contains a pointer or is a character string. This flag must also be used to determine whether the path name delimiter character is 1 or 2 characters long. Valid values follow:

0	The path name is a character string, and the path name delimiter is 1 character long.
1	The path name is a pointer, and the path name delimiter character is 1 character long.
2	The path name is a character string, and the path name delimiter is 2 characters long.
3	The path name is a pointer, and the path name delimiter character is 2 characters long.

Function control block pointer
INPUT; CHAR(*)

A pointer to the data that is passed to **Qp0lProcessSubtree()** on its call. **Qp0lProcessSubtree()** does not process the data that is referred to by this pointer, but passes this pointer as a parameter when it calls the exit program.

Exit program introduced: V4R3

Top | UNIX-Type APIs | APIs by category

Save Storage Free Exit Program

Required Parameter Group:

1	Path name pointers	Input	Char(*)
2	Return code pointer	Output	Binary(4)
3	Return value pointer	Output	Binary(4)
4	Function control block pointer	Input	Char(*)

The Save Storage Free exit program is a user-specified program that is called by **Qp0lSaveStgFree()** to save an i5/OS object of type *STMF. This exit program can be either a procedure or program.

When the Save Storage Free exit program is given control, it should save the object so it can be dynamically retrieved at a later time. The *STMF object is locked when the exit program is called to prevent changes to it until the storage free operation is complete. If the Save Storage Free exit program ends unsuccessfully, it must return a valid *errno* in the storage pointed to by the return value pointer. **Qp0lSaveStgFree()** then passes this *errno* to its caller with a minus one return code.

Storage referred to by the path name pointers or the return code pointer when the Save Storage Free exit program is called is destroyed or reused when **Qp0lSaveStgFree()** regains control.

Authorities and Locks

None.

Required Parameter Group

Path names pointers
INPUT; CHAR(*)

All of the path names to the *STMF object being storage freed. There is one path name for each link to the object. These path names are in the Qlg_Path_Name_T format and are in the UCS-2 CCSID. See Path name format for more information on this format. For information about UCS-2, see the Globalization topic.

Path Name Pointers			
Offset			
Dec	Hex	Type	Field
0	0	BINARY(4)	Number of path names
4	4	CHAR(12)	Reserved
16	10	ARRAY(*)	Array of path name pointers

Array of path name pointers. Pointers to each path name that **Qp0lSaveStgFree()** found for the object identified by the path name on the call to **Qp0lSaveStgFree()**. Each path name is in the Qlg_Path_Name_T format.

Number of path names. The total number of path names that **Qp0lSaveStgFree()** found for the object identified by the caller of **Qp0lSaveStgFree()**.

Reserved. A reserved field. This field must be set to binary zero.

Return code pointer
OUTPUT; BINARY(4)

A pointer to an indicator that is returned to indicate whether the exit program was successful or whether it failed. Valid values follow:

0 The Save Storage Free exit program was successful.
-1 The Save Storage Free exit program was not successful. The Return value pointer is set to indicate the error.

Return value pointer
OUTPUT; BINARY(4)

A pointer to a valid *errno* that is returned from the exit program to identify the reason it was not successful.

Function control block pointer
INPUT; CHAR(*)

A pointer to the data that is passed to **Qp0lSaveStgFree()** on its call. **Qp0lSaveStgFree()** does not process the data that is referred to by this pointer, but passes this pointer as a parameter when it calls the exit program.

Related Information

- "Qp0lSaveStgFree()—Save Storage Free" on page 399—Save Storage Free

Exit program introduced: V4R3

Top | Backup and Recovery APIs | UNIX-Type APIs | APIs by category

Concepts

These are the concepts for this category.

Header Files for UNIX-Type Functions

Programs using the UNIX(R)-type functions must include one or more header files that contain information needed by the functions, such as:

- Macro definitions
- Data type definitions
- Structure definitions
- Function prototypes

The header files are provided in the QSYSINC library, which is optionally installable. Make sure QSYSINC is on your system before compiling programs that use these header files. For information on installing the QSYSINC library, see Include files and the QSYSINC Library.

The table below shows the file and member name in the QSYSINC library for each header file used by the UNIX-type APIs in this publication.

Name of Header File	Name of File in QSYSINC	Name of Member
arpa/inet.h	ARPA	INET
arpa/nameser.h	ARPA	NAMESER
bse.h	H	BSE
bsedos.h	H	BSEDOS
bseerr.h	H	BSEERR
dirent.h	H	DIRENT
errno.h	H	ERRNO
fcntl.h	H	FCNTL
grp.h	H	GRP
inttypes.h	H	INTTYPES
limits.h	H	LIMITS
mman.h	H	MMAN
netdbh.h	H	NETDB
netinet/icmp6.h	NETINET	ICMP6
net/if.h	NET	IF
netinet/in.h	NETINET	IN
netinet/ip_icmp.h	NETINET	IP_ICMP
netinet/ip.h	NETINET	IP
netinet/ip6.h	NETINET	IP6
netinet/tcp.h	NETINET	TCP
netinet/udp.h	NETINET	UDP
netns/idp.h	NETNS	IDP
netns/ipx.h	NETNS	IPX
netns/ns.h	NETNS	NS
netns/sp.h	NETNS	SP

Name of Header File	Name of File in QSYSINC	Name of Member
net/route.h	NET	ROUTE
nettel/tel.h	NETTEL	TEL
os2.h	H	OS2
os2def.h	H	OS2DEF
pwd.h	H	PWD
Qlg.h	H	QLG
qp0lchsg.h	H	QP0LCHSG
qp0lflop.h	H	QP0LFLOP
qp0ljrnl.h	H	QP0LJRNL
qp0lror.h	H	QP0LROR
qp0lrro.h	H	QP0LRRO
qp0lrtsg.h	H	QP0LRTSG
qp0lscan.h	H	QP0LSCAN
Qp0lstdi.h	H	QP0LSTDI
qp0wpid.h	H	QP0WPID
qp0zdipc.h	H	QP0ZDIPC
qp0zipc.h	H	QP0ZIPC
qp0zolip.h	H	QP0ZOLIP
qp0zolsm.h	H	QP0ZOLSM
qp0zripc.h	H	QP0ZRIPC
qp0ztrc.h	H	QP0ZTRC
qp0ztrml.h	H	QP0ZTRML
qp0z1170.h	H	QP0Z1170
qsoasync.h	H	QSOASYNC
qtnxaapi.h	H	QTNXAAPI
qtnxadtp.h	H	QTNXADTP
qtomeapi.h	H	QTOMEAPI
qtossapi.h	H	QTOSSAPI
resolv.h	H	RESOLVE
semaphore.h	H	SEMAPHORE
signal.h	H	SIGNAL
spawn.h	H	SPAWN
ssl.h	H	SSL
sys/errno.h	H	ERRNO
sys/ioctl.h	SYS	IOCTL
sys/ipc.h	SYS	IPC
sys/layout.h	H	LAYOUT
sys/limits.h	H	LIMITS
sys/msg.h	SYS	MSG
sys/param.h	SYS	PARAM
sys/resource.h	SYS	RESOURCE

Name of Header File	Name of File in QSYSINC	Name of Member
sys/sem.h	SYS	SEM
sys/setjmp.h	SYS	SETJMP
sys/shm.h	SYS	SHM
sys/signal.h	SYS	SIGNAL
sys/socket.h	SYS	SOCKET
sys/stat.h	SYS	STAT
sys/statvfs.h	SYS	STATVFS
sys/time.h	SYS	TIME
sys/types.h	SYS	TYPES
sys/uio.h	SYS	UIO
sys/un.h	SYS	UN
sys/wait.h	SYS	WAIT
ulimit.h	H	ULIMIT
unistd.h	H	UNISTD
utime.h	H	UTIME

You can display a header file in QSYSINC by using one of the following methods:

- Using your editor. For example, to display the **unistd.h** header file using the Source Entry Utility editor, enter the following command:
  ```
  STRSEU SRCFILE(QSYSINC/H) SRCMBR(UNISTD) OPTION(5)
  ```
- Using the Display Physical File Member command. For example, to display the **sys/stat.h** header file, enter the following command:
  ```
  DSPPFM FILE(QSYSINC/SYS) MBR(STAT)
  ```

You can print a header file in QSYSINC by using one of the following methods:

- Using your editor. For example, to print the **unistd.h** header file using the Source Entry Utility editor, enter the following command:
  ```
  STRSEU SRCFILE(QSYSINC/H) SRCMBR(UNISTD) OPTION(6)
  ```
- Using the Copy File command. For example, to print the **sys/stat.h** header file, enter the following command:
  ```
  CPYF FROMFILE(QSYSINC/SYS) TOFILE(*PRINT) FROMMBR(STAT)
  ```

Symbolic links to these header files are also provided in directory /QIBM/include.

Top | UNIX-Type APIs | APIs by category

Errno Values for UNIX-Type Functions

Programs using the UNIX(R)-type functions may receive error information as *errno* values. The possible values returned are listed here in ascending *errno* value sequence.

Name	Value	Text	Details
EDOM	3001	A domain error occurred in a math function.	

Name	Value	Text	Details
ERANGE	3002	A range error occurred.	
ETRUNC	3003	Data was truncated on an input, output, or update operation.	
ENOTOPEN	3004	File is not open.	You attempted to do an operation that required the file to be open.
ENOTREAD	3005	File is not opened for read operations.	You tried to read a file that is not open for read operations.
EIO	3006	Input/output error.	» A physical I/O error occurred or a referenced object was damaged. «
ENODEV	3007	No such device.	
ERECIO	3008	Cannot get single character for files opened for record I/O.	The file that was specified is open for record I/O and you attempted to read it as a stream file.
ENOTWRITE	3009	File is not opened for write operations.	You tried to update a file that has not been opened for write operations.
ESTDIN	3010	The stdin stream cannot be opened.	
ESTDOUT	3011	The stdout stream cannot be opened.	
ESTDERR	3012	The stderr stream cannot be opened.	
EBADSEEK	3013	The positioning parameter in fseek is not correct.	
EBADNAME	3014	The object name specified is not correct.	
EBADMODE	3015	The type variable specified on the open function is not correct.	The mode that you attempted to open the file in is not correct.
EBADPOS	3017	The position specifier is not correct.	
ENOPOS	3018	There is no record at the specified position.	You attempted to position to a record that does not exist in the file.
ENUMMBRS	3019	Attempted to use ftell on multiple members.	Remove all but one member from the file.
ENUMRECS	3020	The current record position is too long for ftell.	
EINVAL	3021	The value specified for the argument is not correct.	A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.
EBADFUNC	3022	Function parameter in the signal function is not set.	
ENOENT	3025	No such path or directory.	The directory or a component of the path name specified does not exist.
ENOREC	3026	Record is not found.	
EPERM	3027	The operation is not permitted.	You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.
EBADDATA	3028	Message data is not valid.	The message data that was specified for the error text is not correct.
EBUSY	3029	Resource busy.	An attempt was made to use a system resource that is not available at this time.

Name	Value	Text	Details
EBADOPT	3040	Option specified is not valid.	
ENOTUPD	3041	File is not opened for update operations.	
ENOTDLT	3042	File is not opened for delete operations.	
EPAD	3043	The number of characters written is shorter than the expected record length.	The length of the record is longer than the buffer size that was specified. The data written was padded to the length of the record.
EBADKEYLN	3044	A length that was not valid was specified for the key.	You attempted a record I/O against a keyed file. The key length that was specified is not correct.
EPUTANDGET	3080	A read operation should not immediately follow a write operation.	
EGETANDPUT	3081	A write operation should not immediately follow a read operation.	
EIOERROR	3101	A nonrecoverable I/O error occurred.	
EIORECERR	3102	A recoverable I/O error occurred.	
EACCES	3401	Permission denied.	An attempt was made to access an object in a way forbidden by its object access permissions.
ENOTDIR	3403	Not a directory.	A component of the specified path name existed, but it was not a directory when a directory was expected.
ENOSPC	3404	No space is available.	The requested operations required additional space on the device and there is no space left. This could also be caused by exceeding the user profile storage limit when creating or transferring ownership of an object.
EXDEV	3405	Improper link.	A link to a file on another file system was attempted.
EAGAIN	3406	Operation would have caused the process to be suspended.	
EWOULDBLOCK	3406	Operation would have caused the process to be suspended.	
EINTR	3407	Interrupted function call.	
EFAULT	3408	The address used for an argument was not correct.	In attempting to use an argument in a call, the system detected an address that is not valid.
ETIME	3409	Operation timed out.	
ENXIO	3415	No such device or address.	
EAPAR	3418	Possible APAR condition or hardware failure.	
ERECURSE	3419	Recursive attempt rejected.	
EADDRINUSE	3420	Address already in use.	
EADDRNOTAVAIL	3421	Address is not available.	

Name	Value	Text	Details
EAFNOSUPPORT	3422	The type of socket is not supported in this protocol family.	
EALREADY	3423	Operation is already in progress.	
ECONNABORTED	3424	Connection ended abnormally.	
ECONNREFUSED	3425	A remote host refused an attempted connect operation.	
ECONNRESET	3426	A connection with a remote socket was reset by that socket.	
EDESTADDRREQ	3427	Operation requires destination address.	
EHOSTDOWN	3428	A remote host is not available.	
EHOSTUNREACH	3429	A route to the remote host is not available.	
EINPROGRESS	3430	Operation in progress.	
EISCONN	3431	A connection has already been established.	
EMSGSIZE	3432	Message size is out of range.	
ENETDOWN	3433	The network currently is not available.	
ENETRESET	3434	A socket is connected to a host that is no longer available.	
ENETUNREACH	3435	Cannot reach the destination network.	
ENOBUFS	3436	There is not enough buffer space for the requested operation.	
ENOPROTOOPT	3437	The protocol does not support the specified option.	
ENOTCONN	3438	Requested operation requires a connection.	
ENOTSOCK	3439	The specified descriptor does not reference a socket.	
ENOTSUP	3440	Operation is not supported.	The operation, though supported in general, is not supported for the requested object or the requested arguments.
EOPNOTSUPP	3440	Operation is not supported.	The operation, though supported in general, is not supported for the requested object or the requested arguments.
EPFNOSUPPORT	3441	The socket protocol family is not supported.	
EPROTONOSUPPORT	3442	No protocol of the specified type and domain exists.	
EPROTOTYPE	3443	The socket type or protocols are not compatible.	
ERCVDERR	3444	An error indication was sent by the peer program.	
ESHUTDOWN	3445	Cannot send data after a shutdown.	

Name	Value	Text	Details
ESOCKTNOSUPPORT	3446	The specified socket type is not supported.	
ETIMEDOUT	3447	A remote host did not respond within the timeout period.	
EUNATCH	3448	The protocol required to support the specified address family is not available at this time.	
EBADF	3450	Descriptor is not valid.	A file descriptor argument was out of range, referred to a file that was not open, or a read or write request was made to a file that is not open for that operation.
EMFILE	3452	Too many open files for this process.	An attempt was made to open more files than allowed by the value of OPEN_MAX. The value of OPEN_MAX can be retrieved using the sysconf() function.
ENFILE	3453	Too many open files in the system.	A system limit has been reached for the number of files that are allowed to be concurrently open in the system.
EPIPE	3455	Broken pipe.	
ECANCEL	3456	Operation cancelled.	
EEXIST	3457	Object exists.	The object specified already exists and the specified operation requires that it not exist.
EDEADLK	3459	Resource deadlock avoided.	An attempt was made to lock a system resource that would have resulted in a deadlock situation. The lock was not obtained.
ENOMEM	3460	Storage allocation request failed.	A function needed to allocate storage, but no storage is available.
EOWNERTERM	3462	The synchronization object no longer exists because the owner is no longer running.	The process that had locked the mutex is no longer running, so the mutex was deleted.
EDESTROYED	3463	The synchronization object was destroyed, or the object no longer exists.	
ETERM	3464	Operation was terminated.	
ENOENT1	3465	No such file or directory.	A component of a specified path name did not exist, or the path name was an empty string.
ENOEQFLOG	3466	Object is already linked to a dead directory.	The link as a dead option was specified, but the object is already marked as dead. Only one dead link is allowed for an object.
EEMPTYDIR	3467	Directory is empty.	A directory with entries of only dot and dot-dot was supplied when a nonempty directory was expected.

Name	Value	Text	Details
EMLINK	3468	Maximum link count for a file was exceeded.	An attempt was made to have the link count of a single file exceed LINK_MAX. The value of LINK_MAX can be determined using the pathconf() or the fpathconf() function.
ESPIPE	3469	Seek request is not supported for object.	A seek request was specified for an object that does not support seeking.
ENOSYS	3470	Function not implemented.	An attempt was made to use a function that is not available in this implementation for any object or any arguments.
EISDIR	3471	Specified target is a directory.	The path specified named a directory where a file or object name was expected.
EROFS	3472	Read-only file system.	You have attempted an update operation in a file system that only supports read operations.
EUNKNOWN	3474	Unknown system state.	The operation failed because of an unknown system state. See any messages in the job log and correct any errors that are indicated, then retry the operation.
EITERBAD	3475	Iterator is not valid.	
EITERSTE	3476	Iterator is in wrong state for operation.	
EHRICLSBAD	3477	HRI class is not valid.	
EHRICLBAD	3478	HRI subclass is not valid.	
EHRITYPBAD	3479	HRI type is not valid.	
ENOTAPPL	3480	Data requested is not applicable.	
EHRIREQTYP	3481	HRI request type is not valid.	
EHRINAMEBAD	3482	HRI resource name is not valid.	
EDAMAGE	3484	A damaged object was encountered.	
ELOOP	3485	A loop exists in the symbolic links.	This error is issued if the number of symbolic links encountered is more than POSIX_SYMLOOP (defined in the limits.h header file). Symbolic links are encountered during resolution of the directory or path name.
ENAMETOOLONG	3486	A path name is too long.	A path name is longer than PATH_MAX characters or some component of the name is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using the **pathconf()** function.
ENOLCK	3487	No locks are available.	A system-imposed limit on the number of simultaneous file and record locks was reached, and no more were available at that time.

Name	Value	Text	Details
ENOTEMPTY	3488	Directory is not empty.	You tried to remove a directory that is not empty. A directory cannot contain objects when it is being removed.
ENOSYSRSC	3489	System resources are not available.	
ECONVERT	3490	Conversion error.	One or more characters could not be converted from the source CCSID to the target CCSID.
E2BIG	3491	Argument list is too long.	
EILSEQ	3492	Conversion stopped due to input character that does not belong to the input codeset.	
ETYPE	3493	Object type mismatch.	The type of the object referenced by a descriptor does not match the type specified on the interface.
EBADDIR	3494	Attempted to reference a directory that was not found or was destroyed.	
EBADOBJ	3495	Attempted to reference an object that was not found, was destroyed, or was damaged.	
EIDXINVAL	3496	Data space index used as a directory is not valid.	
ESOFTDAMAGE	3497	Object has soft damage.	
ENOTENROLL	3498	User is not enrolled in system distribution directory.	You attempted to use a function that requires you to be enrolled in the system distribution directory and you are not.
EOFFLINE	3499	Object is suspended.	You have attempted to use an object that has had its data saved and the storage associated with it freed. An attempt to retrieve the object's data failed. The object's data cannot be used until it is successfully restored. The object's data was saved and freed either by saving the object with the STG(*FREE) parameter, or by calling an API.
EROOBJ	3500	Object is read-only.	You have attempted to update an object that can be read only.
EEAHDDSI	3501	Hard damage on extended attribute data space index.	
EEASDDSI	3502	Soft damage on extended attribute data space index.	
EEAHDDS	3503	Hard damage on extended attribute data space.	
EEASDDS	3504	Soft damage on extended attribute data space.	
EEADUPRC	3505	Duplicate extended attribute record.	
ELOCKED	3506	Area being read from or written to is locked.	The read or write of an area conflicts with a lock held by another process.
EFBIG	3507	Object too large.	The size of the object would exceed the system allowed maximum size.

Name	Value	Text	Details
EIDRM	3509	The semaphore, shared memory, or message queue identifier is removed from the system.	
ENOMSG	3510	The queue does not contain a message of the desired type and (msgflg logically ANDed with IPC_NOWAIT).	
EFILECVT	3511	File ID conversion of a directory failed.	To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.
EBADFID	3512	A file ID could not be assigned when linking an object to a directory.	The file ID table is missing or damaged. To recover from this error, run the Reclaim Storage (RCLSTG) command as soon as possible.
ESTALE	3513	File or object handle rejected by server.	
ESRCH	3515	No such process.	
ENOTSIGINIT	3516	Process is not enabled for signals.	An attempt was made to call a signal function under one of the following conditions: • The signal function is being called for a process that is not enabled for asynchronous signals. • The signal function is being called when the system signal controls have not been initialized.
ECHILD	3517	No child process.	
EBADH	3520	Handle is not valid.	
ETOOMANYREFS	3523	The operation would have exceeded the maximum number of references allowed for a descriptor.	
ENOTSAFE	3524	Function is not allowed.	Function is not allowed in a job that is running with multiple threads.
EOVERFLOW	3525	Object is too large to process.	The object's data size exceeds the limit allowed by this function.
EJRNDAMAGE	3526	Journal is damaged.	A journal or all of the journal's attached journal receivers are damaged, or the journal sequence number has exceeded the maximum value allowed. This error occurs during operations that were attempting to send an entry to the journal.
EJRNINACTIVE	3527	Journal is inactive.	The journaling state for the journal is *INACTIVE. This error occurs during operations that were attempting to send an entry to the journal.

Name	Value	Text	Details
EJRNRCVSPC	3528	Journal space or system storage error.	The attached journal receiver does not have space for the entry because the storage limit has been exceeded for the system, the object, the user profile, or the group profile. This error occurs during operations that were attempting to send an entry to the journal.
EJRNRMT	3529	Journal is remote.	The journal is a remote journal. Journal entries cannot be sent to a remote journal. This error occurs during operations that were attempting to send an entry to the journal.
ENEWJRNRCV	3530	New journal receiver is needed.	A new journal receiver must be attached to the journal before entries can be journaled. This error occurs during operations that were attempting to send an entry to the journal.
ENEWJRN	3531	New journal is needed.	The journal was not completely created, or an attempt to delete it did not complete successfully. This error occurs during operations that were attempting to start or end journaling, or were attempting to send an entry to the journal.
EJOURNALED	3532	Object already journaled.	A start journaling operation was attempted on an object that is already being journaled.
EJRNENTTOOLONG	3533	Entry is too large to send.	The journal entry generated by this operation is too large to send to the journal.
EDATALINK	3534	Object is a datalink object.	
ENOTAVAIL	3535	Independent Auxiliary Storage Pool (ASP) is not available.	The independent ASP is in Vary Configuration (VRYCFG) or Reclaim Storage (RCLSTG) processing. To recover from this error, wait until processing has completed for the independent ASP.
ENOTTY	3536	I/O control operation is not appropriate.	
EFBIG2	3540	Attempt to write or truncate file past its sort file size limit.	
ETXTBSY	3543	Text file busy.	An attempt was made to execute an i5/OS PASE program that is currently open for writing, or an attempt has been made to open for writing an i5/OS PASE program that is being executed.
EASPGRPNOTSET	3544	ASP group not set for thread.	
ERESTART	3545	A system call was interrupted and may be restarted.	
ESCANFAILURE	3546	Object had scan failure.	An object has been marked as a scan failure due to processing by an exit program associated with the scan-related integrated file system exit points.

Integrated File System APIs—Time Stamp Updates

Each object (file and directory) has three time values associated with it:

Access Time The time that the data in the object is accessed.
Change Time The time that the attributes of the object are changed.
Modify Time The time that the data in the object is changed.

These values are returned by the **stat()**, **fstat()**, **lstat()**, and **QlgStat()** APIs.

When it is stated that an API sets or updates one of these time values, the value may be "marked for update" by the API rather than actually updated. When a subsequent **stat()**, **fstat()**, **lstat()**, and **QlgStat()** API is called, or the file is closed by all processes, the times that were previously "marked for update" are updated and the update marks are cleared.

The value of these times is measured in seconds since the Epoch. The Epoch is the time 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time. If the system date is set prior to 1970, all time values will be zero. The following table shows which of these times are "marked for update" by each of the APIs.

Time Stamp Updates for Integrated File System APIs

Function	Access	Change	Modify
access	No	No	No
accessx	No	No	No
chdir	No	No	No
chmod	No	Yes	No
chown	No	Yes	No
close	No	No	No
closedir	No	No	No
creat[1] (new file)	Yes	Yes	Yes
creat[1] (parent directory of new file)	No	Yes	Yes
creat[2] (existing file)	No	Yes	Yes
DosSetFileLocks	No	No	No
DosSetRelMaxFH	No	No	No
dup	No	No	No
dup2	No	No	No
faccessx	No	No	No
fchdir	No	No	No
fchmod	No	Yes	No
fchown	No	Yes	No
fclear	No	Yes	Yes
fclear64	No	Yes	Yes
fcntl	No	No	No
fpathconf	No	No	No

Time Stamp Updates for Integrated File System APIs			
Function	**Access**	**Change**	**Modify**
fstat	No	No	No
fstatvfs	No	No	No
fsync	No	No	No
ftruncate	No	Yes	Yes
getcwd	Yes[3]	No	No
getegid	No	No	No
geteuid	No	No	No
getgid	No	No	No
getgrgid	No	No	No
getgrgid_r	No	No	No
getgrnam	No	No	No
getgrnam_r	No	No	No
getgroups	No	No	No
getpwnam	No	No	No
getpwnam_r	No	No	No
getpwuid	No	No	No
getpwuid_r	No	No	No
getuid	No	No	No
givedescriptor	No	No	No
ioctl	No	No	No
lchown	No	Yes	No
link[4] (file)	No	Yes	No
link[4] (parent directory)	No	Yes	Yes
lseek	No	No	No
lstat	No	No	No
mkdir[5] (new directory)	Yes	Yes	Yes
mkdir[5] (parent directory)	No	Yes	Yes
mkfifo[6] (new directory)	Yes	Yes	Yes
mkfifo[6] (parent directory)	No	Yes	Yes
open O_CREAT[7] (new file)	Yes	Yes	Yes
open O_CREAT[7] (parent directory)	No	Yes	Yes
open O_TRUNC[8] (existing file)	No	Yes	Yes
open[9] (existing file)	No	No	No
opendir	No	No	No
pathconf	No	No	No
pread	Yes[14]	No	No
pread64	Yes[14]	No	No
pwrite	No	Yes	Yes
pwrite64	No	Yes	Yes
QlgAccess	No	No	No

Time Stamp Updates for Integrated File System APIs			
Function	**Access**	**Change**	**Modify**
QlgAccessx	No	No	No
QlgChdir	No	No	No
QlgChmod	No	Yes	No
QlgChown	No	Yes	No
QlgCreat[1] (new file)	Yes	Yes	Yes
QlgCreat[1] (parent directory of new file)	No	Yes	Yes
QlgCreat[2] (existing file)	No	Yes	Yes
QlgCvtPathToQSYSObjName	No	No	No
QlgGetAttr	No	Yes	No
QlgGetcwd	Yes[3]	No	No
QlgGetPathFromFileID	Yes[10]	No	No
QlgLchown	No	Yes	No
QlgLink[4] (file)	No	Yes	No
QlgLink[4] (parent directory)	No	Yes	Yes
QlgLstat	No	No	No
QlgMkdir[5] (new directory)	Yes	Yes	Yes
QlgMkdir[5] (parent directory)	No	Yes	Yes
QlgMkfifo[5] (new directory)	Yes	Yes	Yes
QlgMkfifo[5] (parent directory)	No	Yes	Yes
QlgOpen O_CREAT[7] (new file)	Yes	Yes	Yes
QlgOpen O_CREAT[7] (parent directory)	No	Yes	Yes
QlgOpen O_TRUNC[8] (existing file)	No	Yes	Yes
QlgOpen[9] (existing file)	No	No	No
QlgOpendir	No	No	No
QlgPathconf	No	No	No
QlgProcessSubtree	Yes	No	No
QlgReaddir	Yes	No	No
QlgReaddir_r	Yes	No	No
QlgReadlink	Yes	No	No
QlgRenameKeep (parent directories)	No	Yes	Yes
QlgRenameUnlink (parent directories)	No	Yes	Yes
QlgRmdir (parent directory)	No	Yes	Yes
QlgSetAttr	No	Yes	No
QlgStat	No	No	No
QlgStatvfs	No	No	No
QlgSymlink[11] (new link)	Yes	Yes	Yes
QlgSymlink[11] (parent directory)	No	Yes	Yes
QlgUtime[13]	No	Yes	No
QlgUnlink[12] (file)	No	Yes	No
QlgUnlink[12] (parent directory)	No	Yes	Yes

Time Stamp Updates for Integrated File System APIs			
Function	**Access**	**Change**	**Modify**
QP0FPTOS	Yes	No	No
QP0LCHSG	No	No	No
Qp0lCvtPathToQSYSObjName	No	No	No
Qp0lGetAttr	No	Yes	No
Qp0lGetPathFromFileID	Yes[10]	No	No
Qp0lProcessSubtree	Yes	No	No
Qp0lRenameKeep (parent directories)	No	Yes	Yes
Qp0lRenameUnlink (parent directories)	No	Yes	Yes
QP0LROR	No	No	No
QP0LRRO	No	No	No
QP0LRTSG	No	No	No
Qp0lSetAttr	No	Yes	No
qsysetegid()	No	No	No
qsyseteuid()	No	No	No
qsysetgid()	No	No	No
qsysetregid()	No	No	No
qsysetreuid()	No	No	No
qsysetuid()	No	No	No
read	Yes[14]	No	No
readv	Yes[14]	No	No
readdir	Yes	No	No
readdir_r	Yes	No	No
readlink	Yes	No	No
rewinddir	No	No	No
rmdir (parent directory)	No	Yes	Yes
select	No	No	No
stat	No	No	No
statvfs	No	No	No
symlink[11] (new link)	Yes	Yes	Yes
symlink[11] (parent directory)	No	Yes	Yes
sysconf	No	No	No
takedescriptor	No	No	No
umask	No	No	No
unlink[12] (file)	No	Yes	No
unlink[12] (parent directory)	No	Yes	Yes
utime[13]	No	Yes	No
write	No	Yes	Yes
writev	No	Yes	Yes

Time Stamp Updates for Integrated File System APIs			
Function	**Access**	**Change**	**Modify**

Notes:

1. When the file did not previously exist, a successful **creat()** or **QlgCreat()** set the access, change, and modification times for the new file. It also sets the change and modification times of the directory that contains the new file (parent directory).
2. When the file previously existed, a successful **creat()** or **QlgCreat()** sets the change and modification times for the file.
3. The access time of each directory in the absolute path name of the current directory (excluding the current directory itself) is updated.
4. A successful **link()** or **QlgLink()** sets the change time of the file and the change and modification times of the directory that contains the new link (parent directory).
5. A successful **mkdir()** or **QlgMkdir()** sets the access, change, and modification times for the new directory. It also sets the change and modification times of the directory that contains the new directory (parent directory).
6. A successful **mkfifo()** or **QlgMkfifo()** sets the access, change, and modification times for the new FIFO (first-in-first-out) special file. It also sets the change and modification times of the parent directory that contains the new FIFO file.
7. When O_CREAT is specified and the file did not previously exist, a successful **open()** or **QlgOpen()** sets the access, change, and modification times for the new file. It also sets the change and modification times of the directory that contains the new file (parent directory).
8. When O_TRUNC is specified and the file previously existed, a successful **open()** or **QlgOpen()** sets the change and modification times for the file.
9. When O_CREAT and O_TRUNC are not specified, **open()** or **QlgOpen()** does not update any time stamps.
10. A successful **Qp0lGetPathFromFileID()** or **QlgGetPathFromFileID()** sets the access time of each directory in the absolute path name to the file.
11. A successful **symlink()** or **QlgSymlink()** sets the access, change, and modification times for the new symbolic link. It also sets the change and modification times of the directory that contains the new directory (parent directory).
12. A successful **unlink()** or **QlgUnlink()** sets the change and modification times of the directory that contains the file being unlinked (parent directory). If the link count for the file is not zero, the change time for the file is set.
13. A successful **utime()** or **QlgUtime()** sets the access and modify times of the file as specified by the application. The change time of the file is set to the current time.
14. If the read operation was done using a scan descriptor passed to one of the integrated file system scan related exit programs, the Access time is **not** updated. See "Integrated File System Scan on Open Exit Program" on page 523 and "Integrated File System Scan on Close Exit Program" on page 513 for more information.

Top | UNIX-Type APIs | APIs by category

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

```
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY  10504-1785
U.S.A.
```

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

```
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan
```

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

© Copyright IBM Corp. 1998, 2006

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) IBM 2006. Portions of this code are derived from IBM Corp. Sample Programs. (C) Copyright IBM Corp. 1998, 2006. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This Application Programming Interfaces (API) publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i5/OS.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:
Advanced 36
Advanced Function Printing
Advanced Peer-to-Peer Networking
AFP
AIX
AS/400
COBOL/400
CUA
DB2
DB2 Universal Database
Distributed Relational Database Architecture
Domino
DPI
DRDA
eServer
GDDM
IBM
Integrated Language Environment
Intelligent Printer Data Stream
IPDS
i5/OS
iSeries
Lotus Notes
MVS
Netfinity
Net.Data
NetView
Notes
OfficeVision
Operating System/2
Operating System/400
OS/2
OS/400
PartnerWorld
PowerPC
PrintManager
Print Services Facility
RISC System/6000
RPG/400
RS/6000
SAA
SecureWay
System/36
System/370
System/38
System/390
VisualAge
WebSphere
xSeries

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Terms and Conditions

Permissions for the use of these Publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these Publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these Publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these Publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these Publications, or reproduce, distribute or display these Publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the Publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the Publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations. IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED ″AS-IS″ AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE

IBM®

Printed in USA