

Technical Reference: Base Operating System and Extensions, Volume 1



Technical Reference: Base Operating System and Extensions, Volume 1

Note Before using this information and the product it supports, read the information in Appendix C, "Notices," on page 1005.
Seventh Edition (August 2004)

Seventh Edition (August 2004)

This edition applies to AIX 5L Version 5.2 and to all subsequent releases of this product until otherwise indicated in new editions.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to Information Development, Department H6DS-905-6C006, 11501 Burnet Road, Austin, Texas 78758-3493. To send comments electronically, use this commercial Internet address: aix6kpub@austin.ibm.com. Any information that you supply may be used without incurring any obligation to you.

© Copyright International Business Machines Corporation 1994, 2004. All rights reserved. US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Book	κνi
Who Should Use This Book	ΚVİ
Highlighting	
Case-Sensitivity in AIX	κvi
ISO 9000 [°]	
32-Bit and 64-Bit Support for the UNIX98 Specification	vii
Related Publications	Vii
Base Operating System (BOS) Runtime Services (A-P)	. 1
a64l or l64a Subroutine	
l64a_r Subroutine	
abort Subroutine	. 3
abs, div, labs, ldiv, imul_dbl, umul_dbl, llabs, or lldiv Subroutine	4
access, accessx, or faccessx Subroutine	. 5
acct Subroutine	. 8
acl_chg or acl_fchg Subroutine	. 9
acl_get or acl_fget Subroutine	11
acl_put or acl_fput Subroutine	13
acl_set or acl_fset Subroutine	15
acos, acosf, or acosl Subroutine	17
acosh, acoshf, or acoshl Subroutine	18
addssys Subroutine	
adjtime Subroutine	
aio_cancel or aio_cancel64 Subroutine	
aio_error or aio_error64 Subroutine	
aio_fsync Subroutine	
aio_nwait Subroutine	29
aio_nwait_timeout Subroutine	
aio_read or aio_read64 Subroutine	
aio_return or aio_return64 Subroutine	37
aio_suspend or aio_suspend64 Subroutine	
aio_write or aio_write64 Subroutine	
alloc, dealloc, print, read_data, read_regs, symbol_addrs, write_data, and write_regs Subroutine	
arm_end Subroutine	
arm_end Dual Call Subroutine	
arm_getid Subroutine	
arm_getid Dual Call Subroutine	
arm_init Subroutine	58
arm_init Dual Call Subroutine	
arm_start Subroutine	
arm_start Dual Call Subroutine	
arm_stop Subroutine	
arm_stop Dual Call Subroutine	
arm_update Subroutine	
arm_update Dual Call Subroutine	
asinh, asinhf, or asinhl Subroutine	
asinf, asinl, or asin Subroutine	
assert Macro	
atan2f, atan2l, or atan2 Subroutine	
atan, atanf, or atanl Subroutine	
atanh, atanhf, or atanhl Subroutine	
atof atoff Subroutine	
atol or atoll Subroutine	78

audit Subroutine	. 79
	. 81
	. 83
	. 85
······································	. 86
	. 89
	. 90
auditread, auditread_r Subroutines	. 92
auditwrite Subroutine	. 93
authenticate Subroutine	. 94
basename Subroutine	. 96
bcopy, bcmp, bzero or ffs Subroutine	
bessel: j0, j1, jn, y0, y1, or yn Subroutine.	. 98
	. 99
brk or sbrk Subroutine	100
bsearch Subroutine	
btowc Subroutine	
check lock Subroutine.	
_clear_lock Subroutine	
cabs, cabsf, or cabsl Subroutine	_
cacos, cacosf, or cacosl Subroutine	
cacosh, cacoshf, or cacoshl Subroutines	
carg, cargf, or cargl Subroutine	
casin, casinf, or casinl Subroutine	_
casinh, casinfh, or casinlh Subroutine	
catan, catanf, or catanf Subroutine	
catanh, catanhf, or catanhl Subroutine	
catclose Subroutine	
catgets Subroutine.	
catopen Subroutine	
cbrtf, cbrtl, or cbrt Subroutine.	113
ccos, ccosf, or ccosl Subroutine.	
ccosh, ccoshf, or ccoshl Subroutine	
ccsidtocs or cstoccsid Subroutine	
ceil, ceilf, or ceill Subroutine	116
cexp, cexpf, or cexpl Subroutine	117
cfgetospeed, cfsetospeed, cfgetispeed, or cfsetispeed Subroutine	118
cheel or fehaal Subrouting	119
chdir Subroutine	
chmod or fchmod Subroutine	
chown, fchown, lchown, chownx, or fchownx Subroutine	
chpass Subroutine	
chroot Subroutine	
chssys Subroutine	
cimag, cimagf, or cimagl Subroutine	
ckuseracct Subroutine	
ckuserID Subroutine	
class, _class, finite, isnan, or unordered Subroutines	
clock Subroutine	
clog, clogf, or clogl Subroutine	
close Subroutine	
compare_and_swap Subroutine	
compile, step, or advance Subroutine	
confstr Subroutine	
conj, conjf, or conjl Subroutine	
cony Subroutines	140

copysign, copysignf, or copysignl Subroutine	
coredump Subroutine	
cosf, cosl, or cos Subroutine	
cosh, coshf, or coshl Subroutine	
cpow, cpowf, or cpowl Subroutine	
cproj, cprojf, or cprojl Subroutine	
creal, crealf, or creall Subroutine	
crypt, encrypt, or setkey Subroutine	
csid Subroutine	
csin, csinf, or csinl Subroutine	
csinh, csinhf, or csinhl Subroutine	
csqrt, csqrtf, or csqrtl Subroutine	
ctan, ctanf, or ctanl Subroutine	
ctanh, ctanhf, or ctanhl Subroutine	162
ctermid Subroutine	
ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine	
ctime_r, localtime_r, gmtime_r, or asctime_r Subroutine	166
ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl	, or
isascii Subroutines	168
cuserid Subroutine	170
defssys Subroutine	171
delssys Subroutine	171
dirname Subroutine	173
disclaim Subroutine	174
dladdr Subroutine	175
dlclose Subroutine	176
dlerror Subroutine	
dlopen Subroutine	
dlsym Subroutine	180
drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, or srand48 Subroutin	e 181
drem Subroutine	183
_end, _etext, or _edata Identifier	
ecvt, fcvt, or gcvt Subroutine	
EnableCriticalSections, BeginCriticalSection, and EndCriticalSection Subroutine	
erf, erff, or erfl Subroutine	187
erfc, erfcf, or erfcl Subroutine	188
errlog Subroutine	189
errlog_close Subroutine	191
errlog_find_first, errlog_find_next, and errlog_find_sequence Subroutines	
errlog_open Subroutine	
errlog_set_direction Subroutine	195
errlog_write Subroutine	195
exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine	
exit, atexit, _exit, or _Exit Subroutine	
exp, expf, or expl Subroutine	205
exp2, exp2f, or exp2l Subroutine	
expm1, expm1f, or expm1l Subroutine	
fabsf, fabsl, or fabs Subroutine	
fattach Subroutine	
fchdir Subroutine	
fclear or fclear64 Subroutine	
fclose or fflush Subroutine	
fcntl, dup, or dup2 Subroutine	
fdetach Subroutine	
fdim, fdimf, or fdiml Subroutine	

fegetenv or fesetenv Subroutine	. 223
fegetexceptflag or fesetexceptflag Subroutine	. 224
fegetround or fesetround Subroutine	. 225
feholdexcept Subroutine	. 225
fence Subroutine	. 226
feof, ferror, clearerr, or fileno Macro	. 227
feraiseexcept Subroutine	. 228
fetch_and_add Subroutine	. 229
fetch_and_and or fetch_and_or Subroutine	. 230
fetestexcept Subroutine	. 231
feupdateenv Subroutine	. 231
finfo or ffinfo Subroutine	. 232
flockfile, ftrylockfile, funlockfile Subroutine	. 233
floor, floorf, floorl, nearest, trunc, itrunc, or uitrunc Subroutine	. 234
fma, fmaf, or fmal Subroutine	. 237
fmax, fmaxf, or fmaxl Subroutine	. 238
fminf or fminl Subroutine	. 239
fmod, fmodf, or fmodl Subroutine	. 239
fmtmsg Subroutine	. 240
fnmatch Subroutine	. 243
fopen, fopen64, freopen, freopen64 or fdopen Subroutine	. 245
	. 248
fork, f_fork, or vfork Subroutine	. 240 251
fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine	. 252
fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag Subroutine	
fp_cpusync Subroutine	. 254
fp_flush_imprecise Subroutine	. 255
1 1/1- 1/1- /1- /1- /1- /1- /1- /	. 256
fp_iop_snan, fp_iop_infsinf, fp_iop_infdinf, fp_iop_zrdzr, fp_iop_infmzr, fp_iop_invcmp, fp_iop_sqrt,	0==
fp_iop_convert, or fp_iop_vxsoft Subroutines	. 257
fp_raise_xcp Subroutine	. 258
fp_read_rnd or fp_swap_rnd Subroutine	. 259
fp_sh_info, fp_sh_trap_info, or fp_sh_set_stat Subroutine	. 260
fp_trap Subroutine	. 263
fp_trapstate Subroutine	. 265
fpclassify Macro	. 266
fread or fwrite Subroutine	. 267
freehostent Subroutine	. 269
frevoke Subroutine	
frexpf, frexpl, or frexp Subroutine	
fscntl Subroutine	. 271
fseek, fseeko, fseeko64, rewind, ftell, ftello, ftello64, fgetpos, fgetpos64, fsetpos, or fsetpos64	
Subroutine	
fsync or fsync_range Subroutine	
ftok Subroutine	. 277
ftw or ftw64 Subroutine	. 278
fwide Subroutine	. 281
fwprintf, wprintf, swprintf Subroutines	. 281
fwscanf, wscanf, swscanf Subroutines	
gai_strerror Subroutine....................................	
gamma Subroutine	
gencore or coredump Subroutine	
get_malloc_log Subroutine	
get_malloc_log_live Subroutine....................................	
get_speed, set_speed, or reset_speed Subroutines	
getargs Subroutine	
avialus vapivalile	
getaudithostattr IDtohost hostfolD nexthost or putaudithostattr Subroutine	298

	299
	300
	302
	303
	307
getcwd Subroutine	308
	310
	313
getenv Subroutine	313
getevars Subroutine	314
getfsent, getfsspec, getfsfile, getfstype, setfsent, or endfsent Subroutine	316
getgid, getegid or gegidx Subroutine	317
getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine	318
getgrgid_r Subroutine	
getgrnam_r Subroutine	
	322
	325
	326
	328
	329
	332
	333
	335
	336
	337
	339
	340
· ·	341
	343
	344
getpid, getpgrp, or getppid Subroutine	
getportattr or putportattr Subroutine	
getpri Subroutine	
	349
	351
	353
	354
getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit Subroutine	
getrpcent, getrpcbyname, getrpcbynumber, setrpcent, or endrpcent Subroutine	
getrusage, getrusage64, times, or vtimes Subroutine	
getroleattr, nextrole or putroleattr Subroutine	
gets or fgets Subroutine	
gets of rights outstouting the first of the first section of the first s	
getsid Subroutine	
getssys Subroutine	
getsubopt Subroutine	
getsubsvr Subroutine	
getthrds Subroutine	
gettimes Subroduite	
gettimer, settimer, restimer, stime, or time Subroutine	
gettimerid Subroutine	
getttyent, getttynam, setttyent, or endttyent Subroutine	
getuid, geteuid, or getuidx Subroutine	
getuinfo Subroutine	
getuserattr, IDtouser, nextuser, or putuserattr Subroutine	
GetUserAuths Subroutine	აგგ

getuserpw, putuserpw, or putuserpwl getusraclattr, nextusracl or putusracl																							. 389 . 391
getutent, getutid, getutline, pututline,					ent	0	rш	tmi	ona	ame	- S	Sub	rοι	ıtir	ne								393
getvfsent, getvfsbytype, getvfsbynan																							
getwc, fgetwc, or getwchar Subroutin																							
getwo, rigotwo, or getworld cubrouting																							
getws or fgetws Subroutine																							
glob Subroutine																							
glob Subroutine																							
grantpt Subroutine																							
HBA_CloseAdapter Subroutine																							. 405
HBA_FreeLibrary Subroutine																							. 405
HBA_GetAdapterAttributes, HBA_Ge																		•	•	•	•	•	. 400
HBA_GetPortAttributesByWWN St																							407
HBA_GetAdapterName Subroutine																							
HBA_GetFcpPersistentBinding Subro																							
HBA_GetFcpTargetMapping Subrout																							
HBA_GetNumberOfAdapters Subrou																							
HBA_GetPortStatistics Subroutine.																							
HBA_GetRNIDMgmtInfo Subroutine																							
HBA_GetVersion Subroutine																							
HBA_LoadLibrary Subroutine																							
HBA_OpenAdapter Subroutine																							
HBA_RefreshInformation Subroutine																							
HBA_SendCTPassThru Subroutine																							
HBA_SendReadCapacity Subroutine																							. 418
HBA_SendReportLUNs Subroutine																							. 419
HBA_SendRNID Subroutine	•		•	•	•	•	٠	•	٠		•	٠	•	٠	•	٠	•	•	•	•	•	•	. 421
HBA_SendScsiInquiry Subroutine .			٠	٠	٠	٠	٠	٠				٠	•	٠	•	٠	٠	٠		٠	٠		. 422
HBA_SetRNIDMgmtInfo Subroutine																							. 423
hsearch, hcreate, or hdestroy Subro																							. 425
hypot, hypotf, or hypotl Subroutine.																							. 426
iconv Subroutine																							. 427
iconv_close Subroutine																					٠		. 429
iconv_open Subroutine																					٠		. 430
ilogbf, ilogbl, or ilogb Subroutine .																					٠		. 432
imaxabs Subroutine																		•		•			. 433
imaxdiv Subroutine																					٠		. 433
IMAIXMapping Subroutine																							. 434
IMAuxCreate Callback Subroutine .																							. 435
IMAuxDestroy Callback Subroutine																							. 435
IMAuxDraw Callback Subroutine .																							. 436
IMAuxHide Callback Subroutine																							
IMBeep Callback Subroutine																							
IMClose Subroutine																							
IMCreate Subroutine																							
IMDestroy Subroutine																							
IMFilter Subroutine																							. 440
IMFreeKeymap Subroutine																							. 441
IMIndicatorDraw Callback Subroutine																							. 441
IMIndicatorHide Callback Subroutine																							. 442
IMInitialize Subroutine																							. 443
IMInitializeKeymap Subroutine																							. 444
IMIoctl Subroutine																							
IMLookupString Subroutine																							
IMProcess Subroutine																							448

IMProcessAuxiliary Subrouti	ne .																							. 449
IMQueryLanguage Subroutir	ne .																							. 450
IMSimpleMapping Subroutin	е.																							. 45
IMTextCursor Callback Subre	outin	е.																						. 452
IMTextDraw Callback Subrou	utine																							. 453
IMTextHide Callback Subrou	tine																							. 450
IMTextStart Callback Subrou	ıtine																							. 454
inet_aton Subroutine																								. 45
initgroups Subroutine																								. 450
initialize Subroutine																								. 450
insque or remque Subroutine	e																							. 45
install_lwcf_handler Subrout	ine .																							. 458
ioctl, ioctlx, ioctl32, or ioctl32																								. 459
isblank Subroutine																								. 462
isendwin Subroutine																								. 463
isfinite Macro																								. 464
isgreater Macro																								. 464
isgreaterequal Subroutine .																					•			. 46
isinf Subroutine																			•		•			. 460
isless Macro																			•	•	•	•	•	. 460
islessequal Macro																			•	•	•	•	•	. 46
islessgreater Macro																			•	•	•	•	•	. 468
isnormal Macro																				•	•	•	•	. 468
isunordered Macro																				•	•	•	•	. 469
iswalnum, iswalpha, iswcntrl																				viir	· nnai	· r c	· hr	. +0
iswxdigit Subroutine																			13 4	vup	,pci	ι, τ	,	. 470
iswblank Subroutine																			•	•	•	•	•	. 47
iswctype or is_wctype Subro																			•	•	•	•	•	. 47
jcode Subroutines																			•	•	•	•	•	. 473
Japanese conv Subroutines																			•	•	•	•		. 474
Japanese ctype Subroutines																			•	•	•	•	•	. 47
kill or killpg Subroutine																			•	•	•	•	•	. 478
kill of killpy Subtoutine		•	•	•	•	•	•	•	•	•	•		 •	•	•	•	•	•	•	•	•	•	•	. 479
kleenup Subroutine		•	•	•	•	•	•	•	•				 •	•	•	•	•	•	•	•	•	•	•	. 473
knlist Subroutine kpidstate Subroutine															٠	•	٠	•	٠	•	•	•	•	. 482
•														•	•	•	•	•	•	•	•	•	•	. 482
_lazySetErrorHandler Subro														•	•	•	•	•	•	•	•	•	•	_
I3tol or Itol3 Subroutine																		•	٠	•	•	•	•	. 484
LAPI_Addr_get Subroutine																								
LAPI_Addr_set Subroutine																								
LAPI_Address Subroutine.																								
LAPI_Address_init Subroutii																								
LAPI_Address_init64 Subro																								
LAPI_Amsend Subroutine.																								
LAPI_Amsendv Subroutine																								
LAPI_Fence Subroutine .																								
LAPI_Get Subroutine																								
LAPI_Getcntr Subroutine .																								
LAPI_Getv Subroutine																								
LAPI_Gfence Subroutine .																								
LAPI_Init Subroutine																								
LAPI_Msg_string Subroutine																								
LAPI_Msgpoll Subroutine.																								. 52
LAPI_Nopoll_wait Subroutin																								
	ie .																							. 524
LAPI_Probe Subroutine .	ie . 	:											 	:										. 524 . 526
	ie . 	:											 	:										. 524 . 526

LAPI_Putv Subroutine	
LAPI_Qenv Subroutine	
LAPI_Resume_totask Subroutine	. 538
LAPI_Rmw Subroutine	. 539
LAPI_Rmw64 Subroutine	. 543
LAPI_Senv Subroutine	. 546
LAPI_Setcntr Subroutine	. 548
LAPI_Setcntr_wstatus Subroutine	. 550
LAPI_Term Subroutine	. 552
LAPI_Util Subroutine	. 553
LAPI_Waitcntr Subroutine	. 561
LAPI_Xfer Subroutine	. 563
layout_object_create Subroutine	. 576
layout_object_editshape or wcslayout_object_editshape Subroutine	. 577
layout_object_getvalue Subroutine	. 580
layout_object_setvalue Subroutine	. 582
layout_object_shapeboxchars Subroutine	. 584
layout_object_transform or wcslayout_object_transform Subroutine	. 585
layout_object_free Subroutine	. 588
Idahread Subroutine	. 589
Idclose or Idaclose Subroutine	. 590
Idexp, Idexpf, or Idexpl Subroutine	. 591
Idfhread Subroutine	. 592
Idgetname Subroutine	
Idlread, Idlinit, or Idlitem Subroutine	. 596
Idlseek or IdnIseek Subroutine	
Idohseek Subroutine	. 598
Idopen or Idaopen Subroutine	
Idrseek or Idnrseek Subroutine	
ldshread or ldnshread Subroutine	
ldsseek or ldnsseek Subroutine	. 604
Idtbindex Subroutine	. 605
Idtbread Subroutine	. 606
Idtbseek Subroutine	. 607
Igamma, Igammaf, or Igammal Subroutine	
lineout Subroutine	
link Subroutine	. 610
lio listio or lio listio64 Subroutine	. 611
Ilrint, Ilrintf, or Ilrintl Subroutine	. 616
Ilround, Ilroundf, or Ilroundl Subroutine	
load Subroutine	
loadbind Subroutine	. 622
loadquery Subroutine	
localeconv Subroutine	
lockfx, lockf, flock, or lockf64 Subroutine	
log10, log10f, or log10l Subroutine	
log1p, log1pf, or log1pl Subroutine	
log2, log2f, or log2l Subroutine	
logbf, logbl, or logb Subroutine	
log, logf, or logI Subroutine	
loginfailed Subroutine	
loginrestrictions Subroutine	
loginsuccess Subroutine	
Irint, Irintf, or Irintl Subroutine	
Iround, Iroundf, or Iroundl Subroutine	
Isearch or Ifind Subroutine	

Iseek, Ilseek or Iseek64 Subroutine	. 646
lvm_querylv Subroutine	. 647
Ivm_querypv Subroutine	. 651
Ivm_queryvg Subroutine	. 654
lvm_queryvgs Subroutine	. 656
malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, or valloc Subroutine	. 657
madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout,	
omout, fmout, m_out, sdiv, or itom Subroutine	. 661
madvise Subroutine	. 663
makecontext or swapcontext Subroutine	. 664
matherr Subroutine	. 665
MatchAllAuths, , MatchAnyAuths, or MatchAnyAuthsList Subroutine	. 667
mblen Subroutine	. 667
mbrlen Subroutine	. 668
mbrtowc Subroutine	. 669
mbsadvance Subroutine	. 670
mbscat, mbscmp, or mbscpy Subroutine	. 671
mbschr Subroutine	. 672
mbsinit Subroutine	. 673
mbsinvalid Subroutine	. 674
mbslen Subroutine	. 674
mbsncat, mbsncmp, or mbsncpy Subroutine	. 675
mbspbrk Subroutine	. 676
mbsrchr Subroutine	. 677
mbsrtowcs Subroutine	. 678
mbstomb Subroutine	. 679
mbstowcs Subroutine	. 680
mbswidth Subroutine	. 681
mbtowc Subroutine	. 681
memccpy, memchr, memcmp, memcpy, memset or memmove Subroutine	. 683
mincore Subroutine	. 684
mkdir Subroutine	. 685
mknod or mkfifo Subroutine	. 687
mktemp or mkstemp Subroutine	. 689
mmap or mmap64 Subroutine	. 690
mntctl Subroutine	. 694
modf, modff, or modfl Subroutine	. 695
moncontrol Subroutine	. 696
monitor Subroutine	. 697
monstartup Subroutine	. 703
mprotect Subroutine	. 703
	. 700
-	. 707
msem_lock Subroutine	. 709
msem_remove Subroutine	
msem_unlock Subroutine	. 711
msgctl Subroutine	. 712
msgget Subroutine	. 714
msgrcv Subroutine	. 715
msgsnd Subroutine	. 718
msgxrcv Subroutine	. 720
msleep Subroutine	. 722
msync Subroutine	. 723
mt_trce Subroutine	. 724
munmap Subroutine	. 726
mwakeup Subroutine	
nan, nanf, or nanl Subroutine	728

nearbyint, nearbyintf, or near																		
nextafter, nextafterf, nextafte																		
newpass Subroutine			 															. 732
nftw or nftw64 Subroutine .			 . ,															. 734
nl_langinfo Subroutine			 . ,															. 736
nlist, nlist64 Subroutine			 															. 738
ns_addr Subroutine			 															. 739
ns_ntoa Subroutine																		
odm_add_obj Subroutine .																		
odm_change_obj Subroutine																		
odm_close_class Subroutine																		
odm_create_class Subroutin																		
odm_err_msg Subroutine .																		
odm_free_list Subroutine .																		
odm_get_by_id Subroutine																		
odm_get_list Subroutine .																		
odm_get_obj, odm_get_first,																		. 750
odm_initialize Subroutine .																		
odm_lock Subroutine																		
odm_mount_class Subroutin																		
odm_modm_class Subroutine																		
odm_rm_by_id Subroutine.																		
odm_rm_class Subroutine.																		
odm_rm_obj Subroutine .																		
odm_run_method Subroutine																		
odm_set_path Subroutine .																		
odm_set_perms Subroutine																		
odm_terminate Subroutine.																		
odm_unlock Subroutine																		
open, openx, open64, creat,																		. 763
opendir, readdir, telldir, seek																		. 771
passwdexpired Subroutine.																	٠	. 773
pathconf or fpathconf Subrou												٠	٠	٠		٠		. 774
pause Subroutine																		. 777
pcap_close Subroutine												٠			•			. 778
pcap_compile Subroutine .																		. 779
pcap_datalink Subroutine .																		
pcap_dispatch Subroutine .																		
pcap_dump Subroutine																		
pcap_dump_close Subroutin																		
pcap_dump_open Subroutine																		
pcap_file Subroutine																		
pcap_fileno Subroutine																		
pcap_geterr Subroutine																		
pcap_is_swapped Subroutine																		
pcap_lookupdev Subroutine			 															. 786
pcap_lookupnet Subroutine			 															. 787
pcap_loop Subroutine			 															. 788
pcap_major_version Subrout																		
pcap_minor_version Subrout	ine	٠.	 															. 790
pcap_next Subroutine			 															. 791
pcap_open_live Subroutine			 															. 791
pcap_open_offline Subroutin	е		 															
pcap_perror Subroutine			 															. 793
pcap_setfilter Subroutine .																		
ncan snanshot Subroutine																		795

pcap_stats Subroutine		
pcap_strerror Subroutine		
pclose Subroutine		
perfstat_cpu Subroutine		
perfstat_cpu_total Subroutine		
perfstat_disk Subroutine		
perfstat_diskadapter Subroutine		
perfstat_diskpath Subroutine		
perfstat_disk_total Subroutine		
perfstat_memory_total Subroutine		
perfstat_netbuffer Subroutine.............................	 	807
perfstat_netinterface Subroutine		
perfstat_netinterface_total Subroutine	 	810
perfstat_pagingspace Subroutine		
perfstat_protocol Subroutine		
perfstat_reset Subroutine	 	814
perror Subroutine	 	814
pipe Subroutine	 	815
plock Subroutine	 	816
pm_cycles Subroutine	 	818
pm_delete_program Subroutine	 	818
pm_delete_program_group Subroutine	 	819
pm_delete_program_mygroup Subroutine	 	820
pm_delete_program_mythread Subroutine	 	821
pm_delete_program_thread Subroutine		
pm_error Subroutine		
 pm_get_data, pm_get_tdata, pm_get_data_cpu, and pm_get_tdata_cpu Subroutine .		
pm_get_data_group and pm_get_tdata_group Subroutine		
pm_get_data_mygroup or pm_get_tdata_mygroup Subroutine		
pm_get_data_mythread or pm_get_tdata_mythread Subroutine		
pm_get_data_thread or pm_get_tdata_thread Subroutine		
pm_get_program Subroutine		
pm_get_program_group Subroutine		
pm_get_program_mygroup Subroutine		
pm_get_program_mythread Subroutine		
pm_get_program_thread Subroutine		
pm_go_program_amoda oddrodamo		
pm reset data Subroutine	 	839
pm_reset_data_group Subroutine	 	
pm_reset_data_mygroup Subroutine		
pm_reset_data_mythread Subroutine		
pm_reset_data_thread Subroutine		
pm_reset_data_tirread Subroutine		
pm_set_program group Subroutine		
pm_set_program_group Subroutine		
pm_set_program_mythread Subroutine		
. – – 0 – 7		
pm_set_program_thread Subroutine		
pm_start Subroutine		
pm_start_group Subroutine		
pm_start_mygroup Subroutine		
pm_start_mythread Subroutine		
pm_start_thread Subroutine		
pm_stop Subroutine		
pm_stop_group Subroutine		
pm_stop_mygroup Subroutine		
pm_stop_mythread Subroutine	 	859

pm_stop_thread Subroutine	
	. 861
	. 864
_ ' '	. 865
powf, powl, or pow Subroutine	. 866
	. 869
	. 876
psdanger Subroutine	. 878
psignal Subroutine or sys_siglist Vector	. 879
pthdb_attr, pthdb_cond, pthdb_condattr, pthdb_key, pthdb_mutex, pthdb_mutexattr, pthdb_pthread,	076
pthdb_pthread_key, pthdb_rwlock, or pthdb_rwlockattr Subroutine	. 879
pthdb_attr_detachstate,pthdb_attr_addr, pthdb_attr_guardsize,pthdb_attr_inheritsched,	
pthdb_attr_schedparam,pthdb_attr_schedpolicy, pthdb_attr_schedpriority,pthdb_attr_scope,	001
pthdb_attr_stackaddr,pthdb_attr_stacksize, or pthdb_attr_suspendstate Subroutine	. 881
pthdb_condattr_pshared, or pthdb_condattr_addr Subroutine	. 883
pthdb_cond_addr, pthdb_cond_mutex or pthdb_cond_pshared Subroutine	. 884
pthdb_mutexattr_addr, pthdb_mutexattr_prioceiling, pthdb_mutexattr_protocol,	000
pthdb_mutexattr_pshared or pthdb_mutexattr_type Subroutine	. 886
pthdb_mutex_addr, pthdb_mutex_lock_count, pthdb_mutex_owner, pthdb_mutex_pshared,	
pthdb_mutex_prioceiling, pthdb_mutex_protocol, pthdb_mutex_state or pthdb_mutex_type	007
Subroutine	. 887
pthdb_mutex_waiter, pthdb_cond_waiter, pthdb_rwlock_read_waiter or pthdb_rwlock_write_waiter Subroutine	000
	. 889
pthdb_pthread_arg Subroutine	. 890
	. 895
/	. 896
pthdb_pthread_sigmask, pthdb_pthread_sigpend or pthdb_pthread_sigwait Subroutine	. 897
pthdb_pthread_specific Subroutine	. 898
pthdb_pthread_tid or pthdb_tid_pthread Subroutine	. 899
pthdb_rwlockatti_addr, or pthdb_rwlockatti_pshared Subroutine	. 098
pthdb_rwlock_addr, pthdb_rwlock_lock_count, pthdb_rwlock_owner, pthdb_rwlock_pshared or pthdb_rwlock_state Subroutine	. 900
pthdb_session_committed Subroutines	. 900
	. 905
pthread_attr_destroy Subroutine	. 906
pthread_attr_getguardsize or pthread_attr_setguardsize Subroutines	. 907
pthread_attr_getschedparam Subroutine	. 908
pthread_attr_getstackaddr Subroutine	
pthread_attr_getstacksize Subroutine	
pthread_attr_init Subroutine	
pthread_attr_getdetachstate or pthread_attr_setdetachstate Subroutines	913
pthread_attr_getscope and pthread_attr_setscope Subroutines	914
pthread_attr_setschedparam Subroutine	
pthread_attr_setstackaddr Subroutine	
pthread_attr_setstacksize Subroutine	
pthread_attr_setsuspendstate_np and pthread_attr_getsuspendstate_np Subroutine	
pthread_cancel Subroutine	
pthread_cleanup_pop or pthread_cleanup_push Subroutine	920
pthread_cond_destroy or pthread_cond_init Subroutine	921
PTHREAD_COND_INITIALIZER Macro	923
pthread_cond_signal or pthread_cond_broadcast Subroutine	. 924
pthread_cond_wait or pthread_cond_timedwait Subroutine	
pthread_condattr_destroy or pthread_condattr_init Subroutine	
pthread_condattr_getpshared Subroutine	
pthread_condattr_setpshared Subroutine	
othread_create_Subroutine	930

pthread_delay_np Subroutine	32
pthread_equal Subroutine	33
pthread_exit Subroutine	34
pthread_get_expiration_np Subroutine	35
pthread_getconcurrency or pthread_setconcurrency Subroutine	36
pthread_getrusage_np Subroutine	37
pthread_getschedparam Subroutine	10
pthread_getspecific or pthread_setspecific Subroutine	11
pthread_getthrds_np	12
pthread_getunique_np Subroutine..............................9 ²	15
, pthread_join or pthread_detach Subroutine	
pthread_key_create Subroutine	
pthread_key_delete Subroutine	
pthread_kill Subroutine	
pthread_lock_global_np Subroutine	
pthread_mutex_init or pthread_mutex_destroy Subroutine	
PTHREAD_MUTEX_INITIALIZER Macro	
pthread_mutex_lock, pthread_mutex_trylock, or pthread_mutex_unlock Subroutine	
pthread_mutexattr_destroy or pthread_mutexattr_init Subroutine	
pthread_mutexattr_getkind_np Subroutine	
pthread_mutexattr_gettype or pthread_mutexattr_settype Subroutine	
ptirread_mutexattr_setkind_np Subroutine	
pthread_mutexattr_getpshared or pthread_mutexattr_setpshared Subroutine	
pthread_once Subroutine	
PTHREAD_ONCE_INIT Macro	
pthread_rwlock_init or pthread_rwlock_destroy Subroutine	
pthread_rwlock_rdlock or pthread_rwlock_tryrdlock Subroutines	
pthread_rwlock_unlock Subroutine	
pthread_rwlock_wrlock or pthread_rwlock_trywrlock Subroutines	
pthread_rwlockattr_init or pthread_rwlockattr_destroy Subroutines	
pthread_rwlockattr_getpshared or pthread_rwlockattr_setpshared Subroutines	
pthread_self Subroutine	
pthread_setcancelstate, pthread_setcanceltype, or pthread_testcancel Subroutines	
pthread_setschedparam Subroutine	
pthread_sigmask Subroutine	
pthread_signal_to_cancel_np Subroutine	
pthread_suspend_np and pthread_continue_np Subroutine	
pthread_unlock_global_np Subroutine	
pthread_yield Subroutine	
ptrace, ptracex, ptrace64 Subroutine	
ptsname Subroutine	
putc, putchar, fputc, or putw Subroutine	90
putenv Subroutine)2
putgrent Subroutine)3
puts or fputs Subroutine) 4
putwc, putwchar, or fputwc Subroutine) 5
putws or fputws Subroutine	3 7
pwdrestrict_method Subroutine	
Appendix A. Base Operating System Error Codes for Services That Require Path-Name Resolution	
Appendix B. ODM Error Codes	
	٠.

Appendix C.																																
Trademarks.	•	 •	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	٠	•	•	٠	٠	•	•	1006
Index																																1007

About This Book

This book provides information on application programming interfaces to the operating system.

This book is part of the six-volume technical reference set, *AIX 5L Version 5.2 Technical Reference*, that provides information on system calls, kernel extension calls, and subroutines in the following volumes:

- AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1 and AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2 provide information on system calls, subroutines, functions, macros, and statements associated with base operating system runtime services.
- AIX 5L Version 5.2 Technical Reference: Communications Volume 1 and AIX 5L Version 5.2 Technical Reference: Communications Volume 2 provide information on entry points, functions, system calls, subroutines, and operations related to communications services.
- AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1 and AIX 5L Version 5.2
 Technical Reference: Kernel and Subsystems Volume 2 provide information about kernel services,
 device driver operations, file system operations, subroutines, the configuration subsystem, the
 communications subsystem, the low function terminal (LFT) subsystem, the logical volume subsystem,
 the M-audio capture and playback adapter subsystem, the printer subsystem, the SCSI subsystem, and
 the serial DASD subsystem.

This edition supports the release of AIX 5L Version 5.2 with the 5200-04 Recommended Maintenance package. Any specific references to this maintenance package are indicated as AIX 5.2 with 5200-04.

Who Should Use This Book

This book is intended for experienced C programmers. To use the book effectively, you should be familiar with commands, system calls, subroutines, file formats, and special files.

Highlighting

Italics

The following highlighting conventions are used in this book:

Bold Identifies commands, subroutines, keywords, files,

structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects. Identifies parameters whose actual names or values are to

dentines parameters whose actual names of t

be supplied by the user.

Monospace Identifies examples of sp

Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or

information you should actually type.

Case-Sensitivity in AIX

Everything in the AIX operating system is case-sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **Is** command to list files. If you type LS, the system responds that the command is "not found." Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

32-Bit and 64-Bit Support for the UNIX98 Specification

Beginning with Version 4.3, the operating system is designed to support The Open Group's UNIX98 Specification for portability of UNIX-based operating systems. Many new interfaces, and some current ones, have been added or enhanced to meet this specification, making Version 4.3 even more open and portable for applications.

At the same time, compatibility with previous releases of the operating system is preserved. This is accomplished by the creation of a new environment variable, which can be used to set the system environment on a per-system, per-user, or per-process basis.

To determine the proper way to develop a UNIX98-portable application, you may need to refer to The Open Group's UNIX98 Specification, which can be obtained on a CD-ROM by ordering Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification, a book which includes The Open Group's UNIX98 Specification on a CD-ROM.

Related Publications

The following books contain information about or related to application programming interfaces:

- AIX 5L Version 5.2 System Management Guide: Operating System and Devices
- AIX 5L Version 5.2 System Management Guide: Communications and Networks
- · AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs
- AIX 5L Version 5.2 Communications Programming Concepts
- AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts
- AIX 5L Version 5.2 Files Reference

Base Operating System (BOS) Runtime Services (A-P)

a64I or I64a Subroutine

Purpose

Converts between long integers and base-64 ASCII strings.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
long a641 ( String)
char *String;
char *164a ( LongInteger )
long LongInteger;
```

Description

The **a64I** and **I64a** subroutines maintain numbers stored in base-64 ASCII characters. This is a notation in which long integers are represented by up to 6 characters, each character representing a digit in a base-64 notation.

The following characters are used to represent digits:

Character	Description
	Represents 0.
1	Represents 1.
0 -9	Represents the numbers 2-11.
A-Z	Represents the numbers 12-37.
a-z	Represents the numbers 38-63.

Parameters

String Specifies the address of a null-terminated character string.

LongInteger Specifies a long value to convert.

Return Values

The **a64I** subroutine takes a pointer to a null-terminated character string containing a value in base-64 representation and returns the corresponding **long** value. If the string pointed to by the *String* parameter contains more than 6 characters, the **a64I** subroutine uses only the first 6.

Conversely, the **I64a** subroutine takes a **long** parameter and returns a pointer to the corresponding base-64 representation. If the *LongInteger* parameter is a value of 0, the **I64a** subroutine returns a pointer to a null string.

The value returned by the **I64a** subroutine is a pointer into a static buffer, the contents of which are overwritten by each call.

If the *String parameter is a null string, the a64I subroutine returns a value of 0L.

If LongInteger is 0L, the 164a subroutine returns a pointer to a null string.

Related Information

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

List of Multithread Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

164a r Subroutine

Purpose

Converts base-64 long integers to strings.

Library

Thread-Safe C Library (libc_r.a)

Syntax

```
#include <stdlib.h>
```

```
int 164a_r (Convert, Buffer, Length)
long Convert;
char * Buffer;
int Length;
```

Description

The **I64a_r** subroutine converts a given long integer into a base-64 string.

Programs using this subroutine must link to the libpthreads.a library.

For base-64 characters, the following ASCII characters are used:

Character	Description
	Represents 0.
1	Represents 1.
0 -9	Represents the numbers 2-11.
A-Z	Represents the numbers 12-37.
a-z	Represents the numbers 38-63.

The **I64a_r** subroutine places the converted base-64 string in the buffer pointed to by the *Buffer* parameter.

Parameters

Convert Specifies the long integer that is to be converted into a base-64 ASCII string.

Buffer Specifies a working buffer to hold the converted long integer.

Length Specifies the length of the Buffer parameter.

Return Values

- Indicates that the subroutine was successful.
- -1 Indicates that the subroutine was not successful. If the I64a r subroutine is not successful, the errno global variable is set to indicate the error.

Error Codes

If the **I64a** r subroutine is not successful, it returns the following error code:

EINVAL

The Buffer parameter value is invalid or too small to hold the resulting ASCII string.

Related Information

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

List of Multithread Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

abort Subroutine

Purpose

Sends a **SIGIOT** signal to end the current process.

Library

Standard C Library (libc.a)

Syntax

#include <stdlib.h> int abort (void)

Description

The abort subroutine sends a SIGIOT signal to the current process to terminate the process and produce a memory dump. If the signal is caught and the signal handler does not return, the abort subroutine does not produce a memory dump.

If the **SIGIOT** signal is neither caught nor ignored, and if the current directory is writable, the system produces a memory dump in the core file in the current directory and prints an error message.

The abnormal-termination processing includes the effect of the fclose subroutine on all open streams and message-catalog descriptors, and the default actions defined as the SIGIOT signal. The SIGIOT signal is sent in the same manner as that sent by the raise subroutine with the argument SIGIOT.

The status made available to the wait or waitpid subroutine by the abort subroutine is the same as a process terminated by the SIGIOT signal. The abort subroutine overrides blocking or ignoring the SIGIOT signal.

Note: The SIGABRT signal is the same as the SIGIOT signal.

Return Values

The **abort** subroutine does not return a value.

Related Information

The **exit** ("exit, atexit, _exit, or _Exit Subroutine" on page 203), **atexit** ("exit, atexit, _exit, or _Exit Subroutine" on page 203), or _**exit** ("exit, atexit, _exit, or _Exit Subroutine" on page 203) subroutine, **fclose** ("fclose or fflush Subroutine" on page 213) subroutine, **kill** ("kill or killpg Subroutine" on page 478), or **killpg** ("kill or killpg Subroutine" on page 478) subroutine, **raise** subroutine, **sigaction**, **sigvec**, **signal** subroutine, **wait** or **waidtpid** subroutine.

The dbx command.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

abs, div, labs, ldiv, imul_dbl, umul_dbl, llabs, or lldiv Subroutine

Purpose

Computes absolute value, division, and double precision multiplication of integers.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
int abs (i)
int i;
#include <stdlib.h>
long labs (i)
long i;
#include <stdlib.h>
div t div ( Numerator, Denominator)
int Numerator: Denominator;
#include <stdlib.h>
void imul_dbl ( i, j, Result)
long i, j;
long *Result;
#include <stdlib.h>
ldiv t ldiv (Numerator, Denominator)
long Numerator: Denominator;
#include <stdlib.h>
void umul_dbl (i, j, Result)
unsigned long i, j;
unsigned long *Result;
#include <stdlib.h>
long long int llabs(i)
long long int i;
#include <stdlib.h>
11div t 11div (Numerator, Denominator)
long long int Numerator, Denominator;
```

Description

The **abs** subroutine returns the absolute value of its integer operand.

Note: A twos-complement integer can hold a negative number whose absolute value is too large for the integer to hold. When given this largest negative value, the abs subroutine returns the same value.

The div subroutine computes the quotient and remainder of the division of the number represented by the Numerator parameter by that specified by the Denominator parameter. If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented (for example, if the denominator is 0), the behavior is undefined.

The labs and Idiv subroutines are included for compatibility with the ANSI C library, and accept long integers as parameters, rather than as integers.

The **imul_dbl** subroutine computes the product of two signed longs, *i* and *j*, and stores the double long product into an array of two signed longs pointed to by the Result parameter.

The **umul dbl** subroutine computes the product of two unsigned longs, *i* and *j*, and stores the double unsigned long product into an array of two unsigned longs pointed to by the Result parameter.

The **Ilabs** and **Ildiv** subroutines compute the absolute value and division of long long integers. These subroutines operate under the same restrictions as the abs and div subroutines.

Note: When given the largest negative value, the Ilabs subroutine (like the abs subroutine) returns the same value.

Parameters

i	Specifies, for the abs subroutine, some integer; for labs and imul_dbl , some long integer; for the umul_dbl subroutine, some unsigned long integer; for the llabs subroutine, some long long integer.
Numerator	Specifies, for the div subroutine, some integer; for the Idiv subroutine, some long integer; for Ildiv , some long long integer.
j	Specifies, for the imul_dbl subroutine, some long integer; for the umul_dbl subroutine, some unsigned long integer.
Denominator	Specifies, for the div subroutine, some integer; for the Idiv subroutine, some long integer; for Ildiv , some long long integer.
Result	Specifies, for the imul_dbl subroutine, some long integer; for the umul_dbl subroutine, some unsigned long integer.

Return Values

The abs, labs, and llabs subroutines return the absolute value. The imul dbl and umul dbl subroutines have no return values. The div subroutine returns a structure of type div t. The ldiv subroutine returns a structure of type **Idiv** t, comprising the quotient and the remainder. The structure is displayed as:

```
struct ldiv t {
 int quot; /* quotient */
  int rem; /* remainder */
};
```

The **Ildiv** subroutine returns a structure of type **Ildiv_t**, comprising the quotient and the remainder.

access, accessx, or faccessx Subroutine

Purpose

Determines the accessibility of a file.

Library

Standard C Library (libc.a)

Syntax

```
#include <unistd.h>
int access ( PathName, Mode)
char *PathName;
int Mode;
int accessx (PathName, Mode, Who)
char *PathName;
int Mode, Who;
int faccessx ( FileDescriptor, Mode, Who)
int FileDescriptor;
int Mode, Who;
```

Description

The access, accessx, and faccessx subroutines determine the accessibility of a file system object. The accessx and faccessx subroutines allow the specification of a class of users or processes for whom access is to be checked.

The caller must have search permission for all components of the PathName parameter.

Parameters

PathName

FileDescriptor Mode Specifies the path name of the file. If the *PathName* parameter refers to a symbolic link, the **access** subroutine returns information about the file pointed to by the symbolic link. Specifies the file descriptor of an open file.

Specifies the access modes to be checked. This parameter is a bit mask containing 0 or more of the following values, which are defined in the **sys/access.h** file:

R_OK Check read permission.

W OK Check write permission.

X_OK Check execute or search permission.

F_OK Check the existence of a file.

If none of these values are specified, the existence of a file is checked.

Who

Specifies the class of users for whom access is to be checked. This parameter must be one of the following values, which are defined in the sys/access.h file:

ACC_SELF

Determines if access is permitted for the current process. The effective user and group IDs, the concurrent group set and the privilege of the current process are used for the calculation.

ACC_INVOKER

Determines if access is permitted for the invoker of the current process. The real user and group IDs, the concurrent group set, and the privilege of the invoker are used for the calculation.

Note: The expression access (PathName, Mode) is equivalent to accessx (PathName, Mode, ACC_INVOKER).

ACC OTHERS

Determines if the specified access is permitted for any user other than the object owner. The Mode parameter must contain only one of the valid modes. Privilege is not considered in the calculation.

ACC ALL

Determines if the specified access is permitted for all users. The *Mode* parameter must contain only one of the valid modes. Privilege is not considered in the calculation.

Note: The accessx subroutine shows the same behavior by both the user and root with ACC_ALL.

Return Values

If the requested access is permitted, the access, accessx, and faccessx subroutines return a value of 0. If the requested access is not permitted or the function call fails, a value of -1 is returned and the errno global variable is set to indicate the error.

The access subroutine indicates success for X OK even if none of the execute file permission bits are set.

Error Codes

The access and accessx subroutines fail if one or more of the following are true:

EACCES Search permission is denied on a component of the PathName prefix.

EFAULT The PathName parameter points to a location outside the allocated address space of

the process.

ELOOP Too many symbolic links were encountered in translating the PathName parameter. **ENOENT** A component of the PathName does not exist or the process has the disallow

truncation attribute set.

A component of the PathName is not a directory. **ENOTDIR**

ESTALE The process root or current directory is located in a virtual file system that has been

unmounted.

ENOENT The named file does not exist. **ENOENT** The *PathName* parameter was null.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ENAMETOOLONG A component of the PathName parameter exceeded 255 characters or the entire

PathName parameter exceeded 1022 characters.

The **faccessx** subroutine fails if the following is true:

EBADF The value of the FileDescriptor parameter is not valid. The access, accessx, and faccessx subroutines fail if one or more of the following is true:

EIO An I/O error occurred during the operation.

EACCES The file protection does not allow the requested access.

EROFS Write access is requested for a file on a read-only file system.

If Network File System (NFS) is installed on your system, the **accessx** and **faccessx** subroutines can also fail if the following is true:

ETIMEDOUT The connection timed out.

ETXTBSY Write access is requested for a shared text file that is being executed.

EINVAL The value of the *Mode* argument is invalid.

Related Information

The **acl_get** ("acl_get or acl_fget Subroutine" on page 11) subroutine, **chacl** ("chacl or fchacl Subroutine" on page 119) subroutine, **statx** subroutine, **statacl** subroutine.

The aciget command, aciput command, chmod command, chown command.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

acct Subroutine

Purpose

Enables and disables process accounting.

Library

Standard C Library (libc.a)

Syntax

int acct (Path)
char *Path;

Description

The **acct** subroutine enables the accounting routine when the *Path* parameter specifies the path name of the file to which an accounting record is written for each process that terminates. When the *Path* parameter is a 0 or null value, the **acct** subroutine disables the accounting routine.

If the *Path* parameter refers to a symbolic link, the **acct** subroutine causes records to be written to the file pointed to by the symbolic link.

If Network File System (NFS) is installed on your system, the accounting file can reside on another node.

Note: To ensure accurate accounting, each node must have its own accounting file. Although no two nodes should share accounting files, a node's accounting files can be located on any node in the network.

The calling process must have root user authority to use the acct subroutine.

Parameters

Path Specifies a pointer to the path name of the file or a null pointer.

Return Values

Upon successful completion, the acct subroutine returns a value of 0. Otherwise, a value of -1 is returned and the global variable errno is set to indicate the error.

Error Codes

The acct subroutine is unsuccessful if one or more of the following are true:

EACCES Write permission is denied for the named accounting file. **EACCES** The file named by the Path parameter is not an ordinary file.

EBUSY An attempt is made to enable accounting when it is already enabled.

ENOENT The file named by the Path parameter does not exist. **EPERM** The calling process does not have root user authority. **EROFS** The named file resides on a read-only file system.

If NFS is installed on the system, the acct subroutine is unsuccessful if the following is true:

ETIMEDOUT The connection timed out.

acl chg or acl fchg Subroutine

Purpose

Changes the access control information on a file.

Library

Security Library (libc.a)

Syntax

```
#include <sys/access.h>
int acl_chg (Path, How, Mode, Who)
char * Path;
int How;
int Mode;
int Who;
int acl_fchg (FileDescriptor, How, Mode, Who)
int FileDescriptor;
int How;
int Mode;
int Who;
```

Description

The acl_chg and acl_fchg subroutines modify the access control information of a specified file.

Parameters

Mode

Path

FileDescriptor Specifies the file descriptor of an open file.

How Specifies how the permissions are to be altered for the affected entries of the Access

Control List (ACL). This parameter takes one of the following values:

ACC PERMIT

Allows the types of access included in the *Mode* parameter.

ACC DENY

Denies the types of access included in the *Mode* parameter.

ACC SPECIFY

Grants the access modes included in the *Mode* parameter and restricts the

access modes not included in the Mode parameter.

Specifies the access modes to be changed. The Mode parameter is a bit mask containing

zero or more of the following values:

R ACC

Allows read permission.

W ACC

Allows write permission.

X_ACC Allows execute or search permission. Specifies a pointer to the path name of a file.

Who Specifies which entries in the ACL are affected. This parameter takes one of the following

values:

ACC_OBJ_OWNER

Changes the owner entry in the base ACL.

ACC OBJ GROUP

Changes the group entry in the base ACL.

ACC OTHERS

Changes all entries in the ACL except the base entry for the owner.

ACC ALL

Changes all entries in the ACL.

Return Values

On successful completion, the acl_chg and acl_fchg subroutines return a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The acl_chg subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

EACCES Search permission is denied on a component of the Path prefix.

EFAULT The Path parameter points to a location outside of the allocated address space of the

process.

ELOOP Too many symbolic links were encountered in translating the Path parameter. **ENAMETOOLONG**

A component of the Path parameter exceeded 255 characters, or the entire Path

parameter exceeded 1023 characters.

ENOENT A component of the Path does not exist or has the disallow truncation attribute (see

the ulimit subroutine).

ENOENT The Path parameter was null.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

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

The process' root or current directory is located in a virtual file system that has been

unmounted.

The acl fchg subroutine fails and the file permissions remain unchanged if the following is true:

EBADF The FileDescriptor value is not valid.

The acl chg or acl fchg subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

The How parameter is not one of ACC_PERMIT, ACC_DENY, or ACC_SPECIFY. **EINVAL** The Who parameter is not ACC OWNER, ACC GROUP, ACC OTHERS, or ACC ALL. **EINVAL**

EROFS The named file resides on a read-only file system.

The acl_chg or acl_fchg subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

EIO An I/O error occurred during the operation.

EPERM The effective user ID does not match the ID of the owner of the file and the invoker does not

have root user authority.

If Network File System (NFS) is installed on your system, the acl_chg and acl_fchg subroutines can also fail if the following is true:

The connection timed out. **ETIMEDOUT**

Related Information

The acl_get ("acl_get or acl_fget Subroutine") subroutine, acl_put ("acl_put or acl_fput Subroutine" on page 13) subroutine, acl_set ("acl_set or acl_fset Subroutine" on page 15) subroutine, chacl ("chacl or fchacl Subroutine" on page 119) subroutine, chmod ("chmod or fchmod Subroutine" on page 124) subroutine, stat subroutine, statacl subroutine.

The aciget command, aciput command, chmod command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

acl get or acl fget Subroutine

Purpose

Gets the access control information of a file.

Library

Security Library (libc.a)

Syntax

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

```
char *acl get (Path)
char * Path;
```

char *acl_fget (FileDescriptor) int FileDescriptor;

Description

The acl_get and acl_fget subroutines retrieve the access control information for a file system object. This information is returned in a buffer pointed to by the return value. The structure of the data in this buffer is unspecified. The value returned by these subroutines should be used only as an argument to the acl_put or acl_fput subroutines to copy or restore the access control information.

The buffer returned by the acl_get and acl_fget subroutines is in allocated memory. After usage, the caller should deallocate the buffer using the **free** subroutine.

Parameters

Path Specifies the path name of the file.

FileDescriptor Specifies the file descriptor of an open file.

Return Values

On successful completion, the acl_get and acl_fget subroutines return a pointer to the buffer containing the access control information. Otherwise, a null pointer is returned and the errno global variable is set to indicate the error.

Error Codes

The acl_get subroutine fails if one or more of the following are true:

EACCES Search permission is denied on a component of the Path prefix.

EFAULT The Path parameter points to a location outside of the allocated address space of the

process.

ELOOP Too many symbolic links were encountered in translating the *Path* parameter.

ENAMETOOLONG A component of the Path parameter exceeded 255 characters, or the entire Path

parameter exceeded 1023 characters.

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

ENOENT A component of the Path does not exist or the process has the disallow truncation

attribute (see the ulimit subroutine).

ENOENT The Path parameter was null.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ESTALE The process' root or current directory is located in a virtual file system that has been

unmounted.

The acl fget subroutine fails if the following is true:

EBADF The FileDescriptor parameter is not a valid file descriptor.

The acl get or acl fget subroutine fails if the following is true:

EIO An I/O error occurred during the operation.

If Network File System (NFS) is installed on your system, the acl get and acl get subroutines can also fail if the following is true:

ETIMEDOUT The connection timed out.

Security

Access Control The invoker must have search permission for all components of the Path prefix. **Audit Events** None.

Related Information

The acl chg or acl fchg ("acl chg or acl fchg Subroutine" on page 9) subroutine, acl put or acl fput ("acl_put or acl_fput Subroutine") subroutine, acl_set or acl_fset ("acl_set or acl_fset Subroutine" on page 15) subroutine, chacl ("chacl or fchacl Subroutine" on page 119) subroutine, chmod ("chmod or fchmod Subroutine" on page 124) subroutine, **stat** subroutine, **statacl** subroutine.

The aciget command, aciput command, chmod command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

acl_put or acl_fput Subroutine

Purpose

Sets the access control information of a file.

Library

Security Library (libc.a)

Syntax

```
#include <sys/access.h>
int acl put (Path, Access, Free)
char * Path;
char * Access:
int Free;
int acl_fput (FileDescriptor, Access, Free)
int FileDescriptor;
char * Access;
int Free;
```

Description

The acl_put and acl_fput subroutines set the access control information of a file system object. This information is contained in a buffer returned by a call to the acl_get or acl_fget subroutine. The structure of the data in this buffer is unspecified. However, the entire Access Control List (ACL) for a file cannot exceed one memory page (4096 bytes) in size.

Parameters

Path Specifies the path name of a file.

FileDescriptor Specifies the file descriptor of an open file.

Specifies a pointer to the buffer containing the access control information. Access

Free Specifies whether the buffer space is to be deallocated. The following values are valid:

Space is not deallocated.

1 Space is deallocated.

Return Values

On successful completion, the acl_put and acl_fput subroutines return a value of 0. Otherwise, -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The acl put subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

EACCES Search permission is denied on a component of the Path prefix.

EFAULT The Path parameter points to a location outside of the allocated address space of the

process.

ELOOP Too many symbolic links were encountered in translating the *Path* parameter. **ENAMETOOLONG** A component of the Path parameter exceeded 255 characters, or the entire Path

parameter exceeded 1023 characters.

ENOENT A component of the Path does not exist or has the disallow truncation attribute (see

the ulimit subroutine).

ENOENT The Path parameter was null.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

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

ESTALE The process' root or current directory is located in a virtual file system that has been

unmounted.

The acl_fput subroutine fails and the file permissions remain unchanged if the following is true:

EBADF The FileDescriptor parameter is not a valid file descriptor.

The acl put or acl fput subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

EINVAL The Access parameter does not point to a valid access control buffer.

EINVAL The *Free* parameter is not 0 or 1.

FIO An I/O error occurred during the operation.

EROFS The named file resides on a read-only file system.

If Network File System (NFS) is installed on your system, the acl_put and acl_fput subroutines can also fail if the following is true:

ETIMEDOUT The connection timed out.

Security

Access Control: The invoker must have search permission for all components of the Path prefix.

Auditing Events:

Event Information

chacl Path

fchacl FileDescriptor

Related Information

The **acl_chg** ("acl_chg or acl_fchg Subroutine" on page 9) subroutine, **acl_get** ("acl_get or acl_fget Subroutine" on page 11) subroutine, **acl_set** ("acl_set or acl_fset Subroutine") subroutine, **chacl** ("chacl or fchacl Subroutine" on page 119) subroutine, **chmod** ("chmod or fchmod Subroutine" on page 124) subroutine, **stat** subroutine, **statacl** subroutine.

The aciget command, aciput command, chmod command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

acl set or acl fset Subroutine

Purpose

Sets the access control information of a file.

Library

Security Library (libc.a)

Syntax

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

```
int acl_set (Path, OwnerMode, GroupMode, DefaultMode)
char * Path;
int OwnerMode;
int GroupMode;
int DefaultMode;

int acl_fset (FileDescriptor, OwnerMode, GroupMode, DefaultMode)
int * FileDescriptor;
int OwnerMode;
int GroupMode;
int DefaultMode;
```

Description

The acl_set and acl_fset subroutines set the base entries of the Access Control List (ACL) of the file. All other entries are discarded. Other access control attributes are left unchanged.

Parameters

DefaultMode Specifies the access permissions for the default class.

FileDescriptor Specifies the file descriptor of an open file.

GroupMode Specifies the access permissions for the group of the file.

OwnerMode Specifies the access permissions for the owner of the file.

Path Specifies a pointer to the path name of a file.

The mode parameters specify the access permissions in a bit mask containing zero or more of the following values:

R_ACC Authorize read permission. W_ACC Authorize write permission.

X_ACC Authorize execute or search permission.

Return Values

Upon successful completion, the **acl_set** and **acl_fset** subroutines return the value 0. Otherwise, the value -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **acl_set** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

EACCES Search permission is denied on a component of the *Path* prefix.

EFAULT The Path parameter points to a location outside of the allocated address space of the

process.

ELOOP Too many symbolic links were encountered in translating the *Path* parameter. **ENAMETOOLONG** A component of the *Path* parameter exceeded 255 characters, or the entire *Path*

parameter exceeded 1023 characters.

ENOENT A component of the *Path* does not exist or has the **disallow truncation** attribute (see

the ulimit subroutine).

ENOENT The *Path* parameter was null.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

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

ESTALE The process' root or current directory is located in a virtual file system that has been

unmounted.

The acl_fset subroutine fails and the file permissions remain unchanged if the following is true:

EBADF The file descriptor *FileDescriptor* is not valid.

The **acl_set** or **acl_fset** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

EIO An I/O error occurred during the operation.

EPERM The effective user ID does not match the ID of the owner of the file and the invoker does

not have root user authority.

EROFS The named file resides on a read-only file system.

If Network File System (NFS) is installed on your system, the **acl_set** and **acl_fset** subroutines can also fail if the following is true:

ETIMEDOUT The connection timed out.

Security

Access Control: The invoker must have search permission for all components of the Path prefix.

Auditing Events:

Event Information chacl Path

fchacl FileDescriptor

Related Information

The acl_chg ("acl_chg or acl_fchg Subroutine" on page 9) subroutine, acl_get ("acl_get or acl_fget Subroutine" on page 11) subroutine, acl_put ("acl_put or acl_fput Subroutine" on page 13) subroutine, chacl ("chacl or fchacl Subroutine" on page 119) subroutine, chmod ("chmod or fchmod Subroutine" on page 124) subroutine, stat subroutine, statacl subroutine.

The aciget command, aciput command, chmod command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

acos, acosf, or acosl Subroutine

Purpose

Computes the inverse cosine of a given value.

Syntax

```
#include <math.h>
float acosf (x)
float x;
long double acosl (x)
long double x;
double acos (x)
double x;
```

Description

The acosf, acosl, and acos subroutines compute the principal value of the arc cosine of the x parameter. The value of x should be in the range [-1,1].

An application wishing to check for error situations should set the errno global variable to zero and call fetestexcept(FE ALL EXCEPT) before calling these functions. On return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, these subroutines return the arc cosine of x, in the range [0, pi] radians.

For finite values of x not in the range [-1,1], a domain error occurs, and a NaN is returned.

If x is NaN, a NaN is returned.

If x is +1, 0 is returned.

If x is $\pm \ln f$, a domain error occurs, and a NaN is returned.

Related Information

The "acosh, acoshf, or acoshl Subroutine."

math.h in AIX 5L Version 5.2 Files Reference.

acosh, acoshf, or acoshl Subroutine

Purpose

Computes the inverse hyperbolic cosine.

Syntax

```
#include <math.h>
float acoshf (x)
float x;
long double acoshl (X)
long double x;
double acosh (x)
double x;
```

Description

The **acoshf**, or **acoshl** subroutine computes the inverse hyperbolic cosine of the *x* parameter.

The **acosh** subroutine returns the hyperbolic arc cosine specified by the *x* parameter, in the range 1 to the **+HUGE_VAL** value.

An application wishing to check for error situations should set **errno** to zero and call **fetestexcept(FE_ALL_EXCEPT)** before calling these subroutines. Upon return, if the **errno** global variable is nonzero or **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Parameters

x Specifies the value to be computed.

Return Values

Upon successful completion, the **acoshf**, or **acoshl** subroutine returns the inverse hyperbolic cosine of the given argument.

For finite values of x < 1, a domain error occurs, and a NaN is returned.

If x is NaN, a NaN is returned.

If x is +1, 0 is returned.

If x is +Inf, +Inf is returned.

If x is –Inf, a domain error occurs, and a NaN is returned.

Error Codes

The acosh subroutine returns NaNQ (not-a-number) and sets errno to EDOM if the x parameter is less than the value of 1.

Related Information

math.h in AIX 5L Version 5.2 Files Reference.

addssys Subroutine

Purpose

Adds the SRCsubsys record to the subsystem object class.

Library

System Resource Controller Library (libsrc.a)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
int addssys ( SRCSubsystem )
struct SRCsubsys *SRCSubsystem;
```

Description

The addssys subroutine adds a record to the subsystem object class. You must call the defssys subroutine to initialize the SRCSubsystem buffer before your application program uses the SRCsubsys structure. The SRCsubsys structure is defined in the /usr/include/sys/srcobj.h file.

The executable running with this subroutine must be running with the group system.

Parameters

SRCSubsystem A pointer to the SRCsubsys structure.

Return Values

Upon successful completion, the addssys subroutine returns a value of 0. Otherwise, it returns a value of -1 and the **odmerrno** variable is set to indicate the error, or an SRC error code is returned.

Error Codes

The **addssys** subroutine fails if one or more of the following are true:

SRC_BADFSIG Invalid stop force signal. SRC_BADNSIG Invalid stop normal signal. SRC_CMDARG2BIG Command arguments too long.

SRC_GRPNAM2BIG Group name too long.

SRC NOCONTACT Contact not signal, sockets, or message queue.

SRC_NONAME No subsystem name specified. SRC_NOPATH No subsystem path specified. SRC_PATH2BIG Subsystem path too long. SRC_STDERR2BIG stderr path too long. SRC_STDIN2BIG stdin path too long.

SRC_STDOUT2BIG stdout path too long.

SRC_SUBEXIST New subsystem name already on file.

SRC_SUBSYS2BIG Subsystem name too long.

SRC_SYNEXIST New subsystem synonym name already on file.

SRC_SYN2BIG Synonym name too long.

Security

Privilege Control: This command has the Trusted Path attribute. It has the following kernel privilege:

SET_PROC_AUDIT Files Accessed:

Mode File

644 /etc/objrepos/SRCsubsys

Auditing Events:

If the auditing subsystem has been properly configured and is enabled, the **addssys** subroutine generates the following audit record (event) each time the subroutine is executed:

Event Information

SRC_addssys Lists the SRCsubsys records added.

See "Setting Up Auditing" in AIX 5L Version 5.2 Security Guide for details about selecting and grouping audit events, and configuring audit event data collection.

Files

/etc/objrepos/SRCsubsys SRC Subsystem Configuration object class.

/dev/SRC Specifies the AF_UNIX socket file.

/dev/.SRC-unix Specifies the location for temporary socket files.

/usr/include/spc.h Defines external interfaces provided by the SRC subroutines.

/usr/include/sys/srcobj.h Defines object structures used by the SRC.

Related Information

The **chssys** ("chssys Subroutine" on page 132) subroutine, **defssys** ("defssys Subroutine" on page 171) subroutine, **delssys** ("delssys Subroutine" on page 171) subroutine.

The auditpr command, chssys command, mkssys command, rmssys command.

Auditing Overview ("audit Subroutine" on page 79) and System Resource Controller Overview in AIX 5L Version 5.2 System Management Concepts: Operating System and Devices.

Defining Your Subsystem to the SRC, System Resource Controller (SRC) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

List of SRC Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

adjtime Subroutine

Purpose

Corrects the time to allow synchronization of the system clock.

Library

Standard C Library (libc.a)

Syntax

#include <sys/time.h>
int adjtime (Delta, Olddelta)
struct timeval *Delta;
struct timeval *Olddelta;

Description

The **adjtime** subroutine makes small adjustments to the system time, as returned by the **gettimeofday** subroutine, advancing or retarding it by the time specified by the *Delta* parameter of the **timeval** structure. If the *Delta* parameter is negative, the clock is slowed down by incrementing it more slowly than normal until the correction is complete. If the *Delta* parameter is positive, a larger increment than normal is used. The skew used to perform the correction is generally a fraction of one percent. Thus, the time is always a monotonically increasing function, unless the clock is read more than 100 times per second. A time correction from an earlier call to the **adjtime** subroutine may not be finished when the **adjtime** subroutine is called again. If the *Olddelta* parameter is nonzero, then the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call.

This call may be used by time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

The **adjtime** subroutine is restricted to the users with root user authority.

Parameters

Delta Specifies the amount of time to be altered.

Olddelta Contains the number of microseconds still to be corrected from an earlier call.

Return Values

A return value of 0 indicates that the **adjtime** subroutine succeeded. A return value of -1 indicates than an error occurred, and **errno** is set to indicate the error.

Error Codes

The **adjtime** subroutine fails if the following are true:

EFAULT EPERM An argument address referenced invalid memory. The process's effective user ID does not have root user authority.

aio_cancel or aio_cancel64 Subroutine

The **aio_cancel** or **aio_cancel64** subroutine includes information for the POSIX AIO **aio_cancel** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **aio_cancel** subroutine.

POSIX AIO aio cancel Subroutine

Purpose

Cancels one or more outstanding asynchronous I/O requests.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
int aio_cancel (fildes, aiocbp)
int fildes;
struct aiocb *aiocbp;
```

Description

The **aio_cancel** subroutine cancels one or more asynchronous I/O requests currently outstanding against the *fildes* parameter. The *aiocbp* parameter points to the asynchronous I/O control block for a particular request to be canceled. If *aiocbp* is NULL, all outstanding cancelable asynchronous I/O requests against *fildes* are canceled.

Normal asynchronous notification occurs for asynchronous I/O operations that are successfully canceled. If there are requests that cannot be canceled, the normal asynchronous completion process takes place for those requests when they are completed.

For requested operations that are successfully canceled, the associated error status is set to **ECANCELED**, and a -1 is returned. For requested operations that are not successfully canceled, the *aiocbp* parameter is not modified by the **aio_cancel** subroutine.

If *aiocbp* is not NULL, and if *fildes* does not have the same value as the file descriptor with which the asynchronous operation was initiated, unspecified results occur.

The implementation of the subroutine defines which operations are cancelable.

Parameters

fildes Identifies the object to which the outstanding asynchronous I/O requests were originally queued.

aiocbp Points to the **aiocb** structure associated with the I/O operation.

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

```
int aio_fildes
off_t aio_offset
char *aio_buf
size_t aio_nbytes
int aio_reqprio
struct sigevent aio_sigevent
int aio_lio opcode
```

Execution Environment

The aio cancel and aio_cancel64 subroutines can be called from the process environment only.

Return Values

The aio cancel subroutine returns AIO CANCELED to the calling process if the requested operation(s) were canceled. AIO_NOTCANCELED is returned if at least one of the requested operations cannot be canceled because it is in progress. In this case, the state of the other operations, referenced in the call to aio_cancel is not indicated by the return value of aio_cancel. The application may determine the state of affairs for these operations by using the aio_error subroutine. AIO_ALLDONE is returned if all of the operations are completed. Otherwise, the subroutine returns -1 and sets the errno global variable to indicate the error.

Error Codes

EBADF

The *fildes* parameter is not a valid file descriptor.

Related Information

"aio_error or aio_error64 Subroutine" on page 25, "aio_nwait Subroutine" on page 29, "aio_nwait_timeout Subroutine" on page 31, "aio_read or aio_read64 Subroutine" on page 33, "aio_return or aio_return64 Subroutine" on page 37, "aio_suspend or aio_suspend64 Subroutine" on page 40, "aio_write or aio write64 Subroutine" on page 43, and "lio listio or lio listio64 Subroutine" on page 611.

The Asynchronous I/O Subsystem and Communications I/O Subsystem in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

The Input and Output Handling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

Legacy AIO aio_cancel Subroutine

Purpose

Cancels one or more outstanding asynchronous I/O requests.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
```

```
aio cancel (FileDescriptor, aiocbp)
int FileDescriptor:
struct aiocb *aiocbp;
aio_cancel64 (FileDescriptor, aiocbp)
int FileDescriptor;
struct aiocb64 *aiocbp;
```

Description

The aio_cancel subroutine attempts to cancel one or more outstanding asynchronous I/O requests issued on the file associated with the FileDescriptor parameter. If the pointer to the aio control block (aiocb) structure (the aiocbp parameter) is not null, then an attempt is made to cancel the I/O request associated

with this **aiocb**. The *aiocbp* parameter used by the thread calling **aix_cancel** must have had its request initiated by this same thread. Otherwise, a -1 is returned and **errno** is set to EINVAL. However, if the *aiocbp* parameter is null, then an attempt is made to cancel all outstanding asynchronous I/O requests associated with the *FileDescriptor* parameter without regard to the initiating thread.

The **aio_cancel64** subroutine is similar to the **aio_cancel** subroutine except that it attempts to cancel outstanding large file enabled asynchronous I/O requests. Large file enabled asynchronous I/O requests make use of the **aiocb64** structure instead of the aiocb structure. The **aiocb64** structure allows asynchronous I/O requests to specify offsets in excess of OFF_MAX (2 gigbytes minus 1).

In the large file enabled programming environment, aio_cancel is redefined to be aio_cancel64.

When an I/O request is canceled, the **aio_error** ("aio_error or aio_error64 Subroutine" on page 25) subroutine called with the handle to the corresponding **aiocb** structure returns **ECANCELED**.

Note: The _AIO_AIX_SOURCE macro used in **aio.h** must be defined when using **aio.h** to compile an aio application with the Legacy AIO function definitions. The default compile using the **aio.h** file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define _AIO_AIX_SOURCE
#include <sys/aio.h>

or, on the command line when compiling enter:
->xlc ... -D AIO AIX SOURCE ... legacy aio program.c
```

Parameters

FileDescriptor Identifies the object to which the outstanding asynchronous I/O requests were originally queued. aiocbp Points to the **aiocb** structure associated with the I/O operation.

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

```
int aio_whence
off_t aio_offset
char *aio_buf
size_t aio_nbytes
int aio_reqprio
struct event aio_event
struct osigevent aio_event
int aio_flag
aiohandle t aio handle
```

Execution Environment

The aio cancel and aio_cancel64 subroutines can be called from the process environment only.

Return Values

AIO_CANCELED Indicates that all of the asynchronous I/O requests were canceled successfully. The

aio_error subroutine call with the handle to the aiocb structure of the request will return

ECANCELED.

AIO_NOTCANCELED Indicates that the aio_cancel subroutine did not cancel one or more outstanding I/O

requests. This may happen if an I/O request is already in progress. The corresponding error

status of the I/O request is not modified.

AIO_ALLDONE Indicates that none of the I/O requests is in the queue or in progress.

Indicates that the subroutine was not successful. Sets the errno global variable to identify

the error.

-1

A return code can be set to the following **errno** value:

EBADF

Indicates that the *FileDescriptor* parameter is not valid.

Related Information

"aio_error or aio_error64 Subroutine," "aio_nwait Subroutine" on page 29, "aio_nwait_timeout Subroutine" on page 31, "aio_read or aio_read64 Subroutine" on page 33, "aio_return or aio_return64 Subroutine" on page 37, "aio_suspend or aio_suspend64 Subroutine" on page 40, and "aio_write or aio_write64 Subroutine" on page 43, "lio_listio or lio_listio64 Subroutine" on page 611.

The Asynchronous I/O Subsystem and Communications I/O Subsystem in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

The Input and Output Handling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

aio_error or aio_error64 Subroutine

The **aio_error** or **aio_error64** subroutine includes information for the POSIX AIO **aio_error** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **aio error** subroutine.

POSIX AIO aio_error Subroutine

Purpose

Retrieves error status for an asynchronous I/O operation.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
int aio_error (aiocbp)
const struct aiocb *aiocbp;
```

Description

The **aio_error** subroutine returns the error status associated with the **aiocb** structure. This structure is referenced by the *aiocbp* parameter. The error status for an asynchronous I/O operation is the synchronous I/O **errno** value that would be set by the corresponding **read**, **write**, or **fsync** subroutine. If the subroutine has not yet completed, the error status is equal to **EINPROGRESS**.

Parameters

aiocbp Points to the aiocb structure associated with the I/O operation.

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

```
int aio_fildes
off_t aio_offset
char *aio_buf
```

```
size_t aio_nbytes
int
              aio regprio
struct sigevent aio sigevent
               aio_lio_opcode
```

Execution Environment

The aio error and aio error64 subroutines can be called from the process environment only.

Return Values

If the asynchronous I/O operation has completed successfully, the aio error subroutine returns a 0. If unsuccessful, the error status (as described for the read, write, and fsync subroutines) is returned. If the asynchronous I/O operation has not yet completed, **EINPROGRESS** is returned.

Error Codes

EINVAL

The aiocbp parameter does not refer to an asynchronous operation whose return status has not yet been retrieved.

Related Information

"aio_cancel or aio_cancel64 Subroutine" on page 22, "aio_fsync Subroutine" on page 28, "aio_nwait Subroutine" on page 29, "aio nwait timeout Subroutine" on page 31, "aio read or aio read64 Subroutine" on page 33, "aio_return or aio_return64 Subroutine" on page 37, "aio_write or aio_write64 Subroutine" on page 43, "close Subroutine" on page 141, "exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196, "exit, atexit, exit, or Exit Subroutine" on page 203, "fork, f fork, or vfork Subroutine" on page 248, "fsync or fsync range Subroutine" on page 276, "lio listio or lio listio64 Subroutine" on page 611, and "Iseek, Ilseek or Iseek64 Subroutine" on page 646.

read, ready, ready, readvx, or pread Subroutine and write, writey, writey, writevx or pwrite Subroutines in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Legacy AIO aio_error Subroutine

Purpose

Retrieves the error status of an asynchronous I/O request.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
int
aio_error(handle)
aio handle t handle;
int aio error64(handle)
aio_handle_t handle;
```

Description

The aio error subroutine retrieves the error status of the asynchronous request associated with the handle parameter. The error status is the error value that would be set by the corresponding I/O operation. The error status is **EINPROG** if the I/O operation is still in progress.

The aio_error64 subroutine is similar to the aio_error subroutine except that it retrieves the error status associated with an aiocb64 control block.

Note: The _AIO_AIX_SOURCE macro used in **aio.h** must be defined when using **aio.h** to compile an aio application with the Legacy AIO function definitions. The default compile using the **aio.h** file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define _AIO_AIX_SOURCE
#include <sys/aio.h>

or, on the command line when compiling enter:
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Parameters

handle

The handle field of an **aio control block** (**aiocb** or **aiocb64**) structure set by a previous call of the **aio_read**, **aio_read64**, **aio_write**, **aio_write64**, **lio_listio**, **aio_listio64** subroutine. If a random memory location is passed in, random results are returned.

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

int aio whence off t aio offset *aio buf char size t aio nbytes aio reqprio aio event struct event struct osigevent aio_event aio_flag int aiohandle t aio handle

Execution Environment

The aio_error and aio_error64 subroutines can be called from the process environment only.

Return Values

ECANCELED EINPROG Indicates that the operation completed successfully.

Indicates that the I/O request was canceled due to an aio_cancel subroutine call.

Indicates that the I/O request has not completed.

An **errno** value described in the **aio_read** ("aio_read or aio_read64 Subroutine" on page 33), **aio_write** ("aio_write or aio_write64 Subroutine" on page 43), and **lio_listio** ("lio_listio or lio_listio64 Subroutine" on page 611) subroutines: Indicates that the operation was not queued successfully. For example, if the **aio_read** subroutine is called with an unusable file descriptor, it (**aio_read**) returns a value of -1 and sets the **errno** global variable to **EBADF**. A subsequent call of the **aio_error** subroutine with the handle of the unsuccessful **aio control block** (**aiocb**) structure returns **EBADF**.

An **errno** value of the corresponding I/O operation: Indicates that the operation was initiated successfully, but the actual I/O operation was unsuccessful. For example, calling the **aio_write** subroutine on a file located in a full file system returns a value of 0, which indicates the request was queued successfully. However, when the I/O operation is complete (that is, when the **aio_error** subroutine no longer returns **EINPROG**), the **aio_error** subroutine returns **ENOSPC**. This indicates that the I/O was unsuccessful.

Related Information

"aio_cancel or aio_cancel64 Subroutine" on page 22, "aio_read or aio_read64 Subroutine" on page 33, "aio_nwait Subroutine" on page 29, "aio_nwait_timeout Subroutine" on page 31, "aio_return or aio_return64 Subroutine" on page 37, "aio_suspend or aio_suspend64 Subroutine" on page 40, "aio_write or aio_write64 Subroutine" on page 43, "lio_listio or lio_listio64 Subroutine" on page 611, and "lio_listio or lio_listio64 Subroutine" on page 611.

The Asynchronous I/O Overview and the Communications I/O Subsystem: Programming Introduction in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

The Input and Output Handling Programmer's Overview in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs* describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

aio_fsync Subroutine

Purpose

Synchronizes asynchronous files.

Library

Standard C Library (libc.a)

Syntax

#include <aio.h>
int aio_fsync (op, aiocbp)
int op;
struct aiocb *aiocbp;

Description

The **aio_fsync** subroutine asynchronously forces all I/O operations to the synchronized I/O completion state. The function call returns when the synchronization request has been initiated or queued to the file or device (even when the data cannot be synchronized immediately).

If the *op* parameter is set to O_DSYNC, all currently queued I/O operations are completed as if by a call to the **fdatasync** subroutine. If the *op* parameter is set to O_SYNC, all currently queued I/O operations are completed as if by a call to the **fsync** subroutine. If the **aio_fsync** subroutine fails, or if the operation queued by **aio_fsync** fails, outstanding I/O operations are not guaranteed to be completed.

If **aio_fsync** succeeds, it is only the I/O that was queued at the time of the call to **aio_fsync** that is guaranteed to be forced to the relevant completion state. The completion of subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized fashion.

The *aiocbp* parameter refers to an asynchronous I/O control block. The *aiocbp* value can be used as an argument to the *aio_error* and *aio_return* subroutines in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. When the request is queued, the error status for the operation is *EINPROGRESS*. When all data has been successfully transferred, the error status is reset to reflect the success or failure of the operation. If the operation does not complete successfully, the error status for the operation is set to indicate the error. The *aio_sigevent* member determines the asynchronous notification to occur when all operations have achieved synchronized I/O completion. All other members of the structure referenced by the *aiocbp* parameter are ignored. If the control block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, the behavior is undefined.

If the **aio_fsync** subroutine fails or *aiocbp* indicates an error condition, data is not guaranteed to have been successfully transferred.

Parameters

op Determines the way all currently queued I/O operations are completed.

aiocbp Points to the aiocb structure associated with the I/O operation.

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

```
int aio_fildes
off_t aio_offset
char *aio_buf
size_t aio_nbytes
int aio_reqprio
struct sigevent aio_sigevent
int aio_lio opcode
```

Execution Environment

The aio_error and aio_error64 subroutines can be called from the process environment only.

Return Values

The **aio_fsync** subroutine returns a 0 to the calling process if the I/O operation is successfully queued. Otherwise, it returns a -1, and sets the **errno** global variable to indicate the error.

Error Codes

EAGAIN The requested asynchronous operation was not queued due to temporary resource limitations. **EBADF** The *aio_fildes* member of the *aiocb* structure referenced by the *aiocbp* parameter is not a valid

file descriptor open for writing.

In the event that any of the queued I/O operations fail, the **aio_fsync** subroutine returns the error condition defined for the **read** and **write** subroutines. The error is returned in the error status for the asynchronous **fsync** subroutine, which can be retrieved using the **aio error** subroutine.

Related Information

"fcntl, dup, or dup2 Subroutine" on page 214, "fsync or fsync_range Subroutine" on page 276, and "open, open4, creat, or creat64 Subroutine" on page 763.

read, readv, readv, or pread Subroutine and write, writev, writev, writevx or pwrite Subroutines in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

aio nwait Subroutine

Purpose

Suspends the calling process until a certain number of asynchronous I/O requests are completed.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
int aio_nwait (cnt, nwait, list)
int cnt;
int nwait;
struct aiocb **list;
```

Description

Although the **aio_nwait** subroutine is included with POSIX AIO, it is not part of the standard definitions for POSIX AIO.

The **aio_nwait** subroutine suspends the calling process until a certain number (*nwait*) of asynchronous I/O requests are completed. These requests are initiated at an earlier time by the **lio_listio** subroutine, which uses the LIO_NOWAIT_AIOWAIT *cmd* parameter. The **aio_nwait** subroutine fills in the **aiocb** pointers to the completed requests in *list* and returns the number of valid entries in *list*. The *cnt* parameter is the maximum number of **aiocb** pointers that *list* can hold (*cnt* >= *nwait*). The subroutine also returns when less than *nwait* number of requests are done if there are no more pending aio requests.

Note: If the lio_listio64 subroutine is used, the aiocb structure changes to aiocb64.

Note: The aio control block's **errno** field continues to have the value EINPROG until after the **aio_nwait** subroutine is completed. The **aio_nwait** subroutine updates this field when the **lio_listio** subroutine has run with the LIO_NOWAIT_AIOWAIT *cmd* parameter. No utility, such as **aio_error**, can be used to look at this value until after the **aio_nwait** subroutine is completed.

The **aio_suspend** subroutine returns after any one of the specified requests gets done. The **aio_nwait** subroutine returns after a certain number (*nwait* or more) of requests are completed.

There are certain limitations associated with the **aio_nwait** subroutine, and a comparison between it and the **aio suspend** subroutine is necessary. The following table is a comparison of the two subroutines:

aio_suspend:

Requires users to build a list of control blocks, each associated with an I/O operation they want to wait for.

Returns when any one of the specified control blocks indicates that the I/O associated with that control block completed.

The aio control blocks may be updated before the subroutine is called. Other polling methods (such as the **aio_error** subroutine) can also be used to view the aio control blocks.

aio_nwait:

Requires the user to provide an array to put **aiocb** address information into. No specific aio control blocks need to be known.

Returns when *nwait* amount of requests are done or no other requests are to be processed.

Updates the aio control blocks itself when it is called. Other polling methods can't be used until after the <code>aio_nwait</code> subroutine is called enough times to cover all of the aio requests specified with the <code>lio_listio</code> subroutine. Is only used in accordance with the <code>LIO_NOWAIT_AIOWAIT</code> command, which is one of the commands associated with the <code>lio_listio</code> subroutine. If the <code>lio_listio</code> subroutine is not first used with the <code>LIO_NOWAIT_AIOWAIT</code> command, <code>aio_nwait</code> can not be called. The <code>aio_nwait</code> subroutine only affects those requests called by one or more <code>lio_listio</code> calls for a specified process.

Parameters

cnt Specifies the number of entries in the list array.nwait Specifies the minimal number of requests to wait on.

list An array of pointers to all control structures defined in the **alo.h** file.

Return Values

The return value is the total number of requests the aio_nwait subroutine has waited on to complete. It can not be more than cnt. Although nwait is the desired amount of requests to find, the actual amount returned could be less than, equal to, or greater than nwait. The return value indicates how much of the list array to access.

The return value may be greater than the *nwait* value if the **lio_listio** subroutine initiated more than *nwait* requests and the cnt variable is larger than nwait. The nwait parameter represents a minimal value desired for the return value, and *cnt* is the maximum value possible for the return.

The return value may be less than the *nwait* value if some of the requests initiated by the **lio listio** subroutine occur at a time of high activity, and there is a lack of resources available for the number of requests. EAGAIN (error try again later) may be returned in some request's aio control blocks, but these requests will not be seen by the aio nwait subroutine. In this situation aiocb addresses not found on the list have to be accessed by using the aio error subroutine after the aio nwait subroutine is called. You may need to increase the aio parameters max servers or max requests if this occurs. Increasing the parameters will ensure that the system is well tuned, and an EAGAIN error is less likely to occur.

In the event of an error, the aio_nwait subroutine returns a value of -1 and sets the errno global variable to identify the error. Return codes can be set to the following errno values:

EBUSY An aio_nwait call is in process.

EINVAL The application has retrieved all of the aiocb pointers, but the user buffer does not have enough space

EINVAL There are no outstanding async I/O calls.

Related Information

"aio_cancel or aio_cancel64 Subroutine" on page 22, "aio_error or aio_error64 Subroutine" on page 25, "aio_nwait_timeout Subroutine," "aio_read or aio_read64 Subroutine" on page 33, "aio_return or aio_return64 Subroutine" on page 37, "aio_suspend or aio_suspend64 Subroutine" on page 40, "aio_write or aio_write64 Subroutine" on page 43, and "lio_listio or lio_listio64 Subroutine" on page 611.

The Asynchronous I/O Overview and the Communications I/O Subsystem: Programming Introduction in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

The Input and Output Handling Programmer's Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

aio_nwait_timeout Subroutine

Purpose

Extends the capabilities of the aio_nwait subroutine by specifying timeout values.

Library

The aio_nwait_timeout subroutine is not available through the libc.a library or the /usr/include/aio.h file. It must be accessed directly from the kernel.

Syntax

```
int aio_nwait_timeout (cnt, nwait, list, timeout)
int cnt;
int nwait;
struct aiocbp **list;
uint timeout;
```

Description

The aio nwait timeout subroutine is exported from the AIO subsystem.

The **aio_nwait_timeout** subroutine waits for a certain number of asynchronous I/O operations to complete as specified by the *cnt* parameter, or until the call has blocked for a certain duration specified by the *timeout* parameter.

Note: The **aio_nwait_timeout** subroutine has to be redefined and accessed from POSIX AIO as **_posix_aio_nwait_timeout**.

To access the subroutine in Legacy AIO when compiling, use the **aio_nwait_timeout.imp** import file, which contains the following:

```
#!/unix
aio nwait timeout
```

Assuming the **aio_nwait_timout** subroutine is called inside of the **testcase.c** file, compile it with the following:

```
cc -D_AIO_AIX_SOURCE -o testcase -bI:./aio_nwait_timeout.imp testcase.c
```

To access the **aio_nwait_timout** subroutine when compiling in POSIX AIO, use the **aio_nwait_timeout.imp** import file which contains the following:

```
#!/unix
posix aio nwait timeout
```

Assuming the **_posix_aio_nwait_timeout** subroutine is called inside of the **testcase.c** file, compile it with the following:

```
cc -o testcase -bI:./aio_nwait_timeout.imp testcase.c
```

Parameters

cnt Indicates the maximum number of pointers to the aiocbp structure that can be copied into the list array.
 list An array of pointers to aio control structures defined in the aio.h file.

nwait Specifies the number of asynchronous I/O operations that must complete before the aio_nwait_timout subroutine is called.

timeout Specified in units of milliseconds.

A *timeout* value of -1 indicates that the subroutine behaves like the **aio_nwait** subroutine, blocking until all of the requested I/O operations complete or until there are no more asynchronous I/O requests pending from the process.

A *timeout* value of 0 indicates that the subroutine returns immediately with the current completed number of asynchronous I/O requests. All other positive *timeout* values indicate that the subroutine must block until either the *timeout* value is reached or the requested number of asynchronous I/O operations complete.

Return Values

The return value is the total number of requests the **aio_nwait** subroutine has waited on to complete. It can not be more than *cnt*. Although *nwait* is the desired amount of requests to find, the actual amount returned could be less than, equal to, or greater than *nwait*. The return value indicates how much of the list array to access.

The return value may be greater than the *nwait* value if the **lio_listio** subroutine initiated more than *nwait* requests and the *cnt* variable is larger than *nwait*. The *nwait* parameter represents a minimal value desired for the return value, and *cnt* is the maximum value possible for the return.

The return value may be less than the *nwait* value if some of the requests initiated by the **lio_listio** subroutine occur at a time of high activity, and there is a lack of resources available for the number of requests. The **EAGAIN** return code (error try again later) might be returned in some request's aio control blocks, but these requests will not be seen by the **aio_nwait** subroutine. In this situation, the **aiocb** structure addresses that are not found on the list must be accessed using the **aio_error** subroutine after the **aio_nwait** subroutine is called. You might need to increase the aio parameters max servers or max requests if this occurs. Increasing the parameters will ensure that the system is well tuned, and an **EAGAIN** error is less likely to occur. The return value might be less than the *nwait* value due to the setting of the new timeout parameter in the following cases:

- timeout > 0 and a timeout has occurred before nwait requests are done
- timeout = 0 and the current requests completed at the time of the aio_nwait_timeout call are less then
 nwait parameter

In the event of an error, the **aio_nwait** subroutine returns a value of -1 and sets the **errno** global variable to identify the error. Return codes can be set to the following **errno** values:

EBUSY An aio_nwait call is in process.

EINVAL The application has retrieved all of the **aiocb** pointers, but the user buffer does not have enough space

for them.

EINVAL There are no outstanding async I/O calls.

Related Information

"aio_nwait Subroutine" on page 29, "aio_suspend or aio_suspend64 Subroutine" on page 40, "aio_cancel or aio_cancel64 Subroutine" on page 22, "aio_error or aio_error64 Subroutine" on page 25, "aio_read or aio_read64 Subroutine," "aio_return or aio_return64 Subroutine" on page 37, "aio_write or aio_write64 Subroutine" on page 43, and "lio_listio or lio_listio64 Subroutine" on page 611.

The Asynchronous I/O Overview and the Communications I/O Subsystem: Programming Introduction in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

The Input and Output Handling Programmer's Overview in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs* describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

aio read or aio read64 Subroutine

The aio_read or aio_read64 subroutine includes information for the POSIX AIO aio_read subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO aio read subroutine.

POSIX AIO aio read Subroutine

Purpose

Asynchronously reads a file.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
int aio read (aiocbp)
struct aiocb *aiocbp;
```

Description

The aio read subroutine reads aio nbytes from the file associated with aio fildes into the buffer pointed to by aio_buf. The subroutine returns when the read request has been initiated or queued to the file or device (even when the data cannot be delivered immediately).

The aiocbp value may be used as an argument to the aio error and aio return subroutines in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. If an error condition is encountered during queuing, the function call returns without having initiated or queued the request. The requested operation takes place at the absolute position in the file as given by aio_offset, as if the Iseek subroutine were called immediately prior to the operation with an offset equal to aio offset and a whence equal to SEEK SET. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The aio lio opcode field is ignored by the aio read subroutine.

Since prioritized I/O is not supported at this time the aio_reaprio field of the aiocb structure is not presently used.

The aiocbp parameter points to an aiocb structure. If the buffer pointed to by aio_buf or the control block pointed to by aiocbp becomes an illegal address prior to asynchronous I/O completion, the behavior is undefined.

Simultaneous asynchronous operations using the same aiocbp produce undefined results.

If synchronized I/O is enabled on the file associated with aio_fildes, the behavior of this subroutine is according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding. the result of that action is undefined.

For regular files, no data transfer occurs past the offset maximum established in the open file description.

Parameters

Points to the aiocb structure associated with the I/O operation. aiocbp

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

```
int
               aio fildes
off t
               aio offset
char
             *aio buf
size_t
             aio_nbytes
               aio_reqprio
int
struct sigevent aio sigevent
               aio lio opcode
```

Execution Environment

The aio_read and aio_read64 subroutines can be called from the process environment only.

Return Values

The **aio_read** subroutine returns 0 to the calling process if the I/O operation is successfully queued. Otherwise, it returns a -1 and sets the **errno** global variable to indicate the error.

Error Codes

EAGAIN The requested asynchronous I/O operation was not gueued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to the **aio_read** subroutine, or asynchronously. If any of the conditions below are detected synchronously, the **aio_read** subroutine returns -1 and sets the **errno** global variable to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation is set to the corresponding value.

EBADF The *aio_fildes* parameter is not a valid file descriptor open for reading.

EINVAL The file offset value implied by *aio_offset* is invalid, *aio_reqprio* is an invalid value, or *aio_nbytes* is

an invalid value.

If the aio_read subroutine successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is one of the values normally returned by the **read** subroutine. In addition, the error status of the asynchronous operation is set to one of the error statuses normally set by the **read** subroutine, or one of the following values:

EBADF The *aio_fildes* argument is not a valid file descriptor open for reading.

ECANCELED The requested I/O was canceled before the I/O completed due to an explicit aio_cancel request.

EINVAL The file offset value implied by *aio_offset* is invalid.

The following condition may be detected synchronously or asynchronously:

EOVERFLOW The file is a regular file, *aio_nbytes* is greater than 0, and the starting offset in *aio_offset* is before the end-of-file and is at or beyond the offset maximum in the open file description associated with

aio_fildes.

Related Information

"aio_cancel or aio_cancel64 Subroutine" on page 22, "aio_error or aio_error64 Subroutine" on page 25, "aio_nwait Subroutine" on page 29, "aio_nwait_timeout Subroutine" on page 31, "aio_return or aio_return64 Subroutine" on page 37, "aio_suspend or aio_suspend64 Subroutine" on page 40, "aio_write or aio_write64 Subroutine" on page 43, "close Subroutine" on page 141, "exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196, "exit, atexit, _exit, or _Exit Subroutine" on page 203, "fork, f_fork, or vfork Subroutine" on page 248, "lio_listio or lio_listio64 Subroutine" on page 611, and "lseek, llseek or lseek64 Subroutine" on page 646.

The read, readx, readv, readvx, or pread Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Legacy AIO aio_read Subroutine

Purpose

Reads asynchronously from a file.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
int aio read( FileDescriptor, aiocbp)
int FileDescriptor;
struct aiocb *aiocbp;
int aio read64( FileDescriptor, aiocbp)
int FileDescriptor;
struct aiocb64 *aiocbp;
```

Description

The aio_read subroutine reads asynchronously from a file. Specifically, the aio_read subroutine reads from the file associated with the *FileDescriptor* parameter into a buffer.

The aio_read64 subroutine is similar to the aio_read subroutine execpt that it takes an aiocb64 reference parameter. This allows the aio read64 subroutine to specify offsets in excess of OFF MAX (2 gigbytes minus 1).

In the large file enabled programming environment, aio read is redefined to be aio read64.

The details of the read are provided by information in the aiocb structure, which is pointed to by the aiocbp parameter. This information includes the following fields:

aio buf Indicates the buffer to use.

aio nbytes Indicates the number of bytes to read.

When the read request has been queued, the aio read subroutine updates the file pointer specified by the aio whence and aio offset fields in the aiocb structure as if the requested I/O were already completed. It then returns to the calling program. The aio whence and aio offset fields have the same meaning as the whence and offset parameters in the Iseek ("Iseek, Ilseek or Iseek64 Subroutine" on page 646) subroutine. The subroutine ignores them for file objects that are not capable of seeking.

If an error occurs during the call, the read request is not gueued. To determine the status of a request, use the aio_error ("aio_error or aio_error64 Subroutine" on page 25) subroutine.

To have the calling process receive the SIGIO signal when the I/O operation completes, set the AIO_SIGNAL bit in the aio flag field in the aiocb structure.

Note: The event structure in the aiocb structure is currently not in use but is included for future compatibility.

Note: The _AIO_AIX_SOURCE macro used in aio.h must be defined when using aio.h to compile an aio application with the Legacy AIO function definitions. The default compile using the aio.h file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
or, on the command line when compiling enter:
->xlc ... -D AIO AIX SOURCE ... legacy aio program.c
```

Parameters

FileDescriptor Identifies the object to be read as returned from a call to open.

aiocbp Points to the asynchronous I/O control block structure associated with the I/O operation.

aiocb Structure

The aiocb and the aiocb64 structures are defined in the aio.h file and contains the following members:

aio whence off t aio offset char *aio_buf aio nbytes size t aio regprio int struct event aio event struct osigevent aio_event int aio flag aio_handle aiohandle t

Execution Environment

The aio read and aio read64 subroutines can be called from the process environment only.

Return Values

When the read request queues successfully, the aio_read subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the global variable errno to identify the error.

Return codes can be set to the following **errno** values:

EAGAIN Indicates that the system resources required to gueue the request are not available. Specifically, the

transmit queue may be full, or the maximum number of opens may be reached.

EBADF Indicates that the FileDescriptor parameter is not valid.

EFAULT Indicates that the address specified by the *aiocbp* parameter is not valid.

EINVAL Indicates that the aio whence field does not have a valid value, or that the resulting pointer is not valid.

Note: Other error codes defined in the sys/errno.h file can be returned by aio_error if an error during the I/O operation is encountered.

Related Information

The "aio cancel or aio cancel64 Subroutine" on page 22, "aio error or aio error64 Subroutine" on page 25, "aio nwait Subroutine" on page 29, "aio nwait timeout Subroutine" on page 31, "aio suspend or aio_suspend64 Subroutine" on page 40, "aio_write or aio_write64 Subroutine" on page 43, "aio_return or aio_return64 Subroutine," and "lio_listio or lio_listio64 Subroutine" on page 611.

The Asynchronous I/O Overview and the Communications I/O Subsystem: Programming Introduction in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

The Input and Output Handling Programmer's Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

aio return or aio_return64 Subroutine

The aio return or aio return64 subroutine includes information for the POSIX AIO aio return subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO aio return subroutine.

POSIX AIO aio return Subroutine

Purpose

Retrieves the return status of an asynchronous I/O operation.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
size t aio return (aiocbp);
struct aiocb *aiocbp;
```

Description

The aio_return subroutine returns the return status associated with the aiocb structure. The return status for an asynchronous I/O operation is the value that would be returned by the corresponding read, write, or fsync subroutine call. If the error status for the operation is equal to EINPROGRESS, the return status for the operation is undefined. The aio return subroutine can be called once to retrieve the return status of a given asynchronous operation. After that, if the same aiocb structure is used in a call to aio return or aio_error, an error may be returned. When the aiocb structure referred to by aiocbp is used to submit another asynchronous operation, the aio return subroutine can be successfully used to retrieve the return status of that operation.

Parameters

aiocbp

Points to the **aiocb** structure associated with the I/O operation.

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

```
aio fildes
off t
                aio offset
char
               *aio_buf
size_t
               aio nbytes
int
                aio regprio
struct sigevent aio sigevent
                aio lio opcode
```

Execution Environment

The aio_return and aio_return64 subroutines can be called from the process environment only.

Return Values

If the asynchronous I/O operation has completed, the return status (as described for the read, write, and fsync subroutines) is returned. If the asynchronous I/O operation has not yet completed, the results of the aio return subroutine are undefined.

Error Codes

EINVAL

The aiocbp parameter does not refer to an asynchronous operation whose return status has not yet been retrieved.

Related Information

"aio_cancel or aio_cancel64 Subroutine" on page 22, "aio_error or aio_error64 Subroutine" on page 25, "aio nwait Subroutine" on page 29, "aio nwait timeout Subroutine" on page 31, "aio read or aio read64 Subroutine" on page 33, "aio_suspend or aio_suspend64 Subroutine" on page 40, "aio_write or aio_write64 Subroutine" on page 43, "close Subroutine" on page 141, "exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196, "exit, atexit, _exit, or _Exit Subroutine" on page 203, "fork, f_fork, or vfork Subroutine" on page 248, "lio_listio or lio_listio64 Subroutine" on page 611, and "lseek, llseek or lseek64 Subroutine" on page 646.

The read, ready, ready, or pread Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Legacy AIO aio_return Subroutine

Purpose

Retrieves the return status of an asynchronous I/O request.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
int aio_return( handle)
aio_handle_t handle;
int aio_return64( handle)
aio_handle_t handle;
```

Description

The aio_return subroutine retrieves the return status of the asynchronous I/O request associated with the aio_handle_t handle if the I/O request has completed. The status returned is the same as the status that would be returned by the corresponding read or write function calls. If the I/O operation has not completed, the returned status is undefined.

The **aio_return64** subroutine is similar to the **aio_return** subroutine except that it retrieves the error status associated with an **aiocb64** control block.

Note: The _AIO_AIX_SOURCE macro used in **aio.h** must be defined when using **aio.h** to compile an aio application with the Legacy AIO function definitions. The default compile using the **aio.h** file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define _AIO_AIX_SOURCE
#include <sys/aio.h>

or, on the command line when compiling enter:
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Parameters

handle

The handle field of an **aio control block** (**aiocb** or **aiocb64**) structure is set by a previous call of the **aio_read**, **aio_read64**, **aio_write**, **aio_write64**, **lio_listio**, **aio_listio64** subroutine. If a random memory location is passed in, random results are returned.

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

```
int aio_whence
off_t aio_offset
char *aio_buf
size_t aio_nbytes
int aio_reqprio
struct event aio_event
struct osigevent aio_event
int aio_flag
aiohandle t aio_handle
```

Execution Environment

The aio_return and aio_return64 subroutines can be called from the process environment only.

Return Values

The **aio_return** subroutine returns the status of an asynchronous I/O request corresponding to those returned by **read** or **write** functions. If the error status returned by the **aio_error** subroutine call is **EINPROG**, the value returned by the **aio return** subroutine is undefined.

Examples

An **aio_read** request to read 1000 bytes from a disk device eventually, when the **aio_error** subroutine returns a 0, causes the **aio_return** subroutine to return 1000. An **aio_read** request to read 1000 bytes from a 500 byte file eventually causes the **aio_return** subroutine to return 500. An **aio_write** request to write to a read-only file system results in the **aio_error** subroutine eventually returning **EROFS** and the **aio_return** subroutine returning a value of -1.

Related Information

"aio_cancel or aio_cancel64 Subroutine" on page 22, "aio_error or aio_error64 Subroutine" on page 25, "aio_nwait Subroutine" on page 29, "aio_nwait_timeout Subroutine" on page 31, "aio_read or aio_read64 Subroutine" on page 33, "aio_suspend or aio_suspend64 Subroutine," "aio_write or aio_write64 Subroutine" on page 43, "close Subroutine" on page 141, "exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196, "exit, atexit, _exit, or _Exit Subroutine" on page 203, "fork, f_fork, or vfork Subroutine" on page 248, "lio_listio or lio_listio64 Subroutine" on page 611, and "lseek, llseek or lseek64 Subroutine" on page 646.

The Asynchronous I/O Overview and the Communications I/O Subsystem: Programming Introduction in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

The Input and Output Handling Programmer's Overview in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs* describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

aio_suspend or aio_suspend64 Subroutine

The **aio_suspend** subroutine includes information for the POSIX AIO **aio_suspend** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **aio suspend** subroutine.

POSIX AIO aio suspend Subroutine

Purpose

Waits for an asynchronous I/O request.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
int aio suspend (list, nent,
timeout)
const struct aiocb * const list[];
int nent;
const struct timespec *timeout;
```

Description

The aio_suspend subroutine suspends the calling thread until at least one of the asynchronous I/O operations referenced by the list parameter has completed, until a signal interrupts the function, or, if timeout is not NULL, until the time interval specified by timeout has passed. If any of the aiocb structures in the list correspond to completed asynchronous I/O operations (the error status for the operation is not equal to EINPROGRESS) at the time of the call, the subroutine returns without suspending the calling thread. The *list* parameter is an array of pointers to asynchronous I/O control blocks. The *nent* parameter indicates the number of elements in the array. Each aiocb structure pointed to has been used in initiating an asynchronous I/O request through the aio_read, aio_write, or lio_listio subroutine. This array may contain NULL pointers, which are ignored. If this array contains pointers that refer to aiocb structures that have not been used in submitting asynchronous I/O, the effect is undefined.

If the time interval indicated in the **timespec** structure pointed to by *timeout* passes before any of the I/O operations referenced by list are completed, the aio suspend subroutine returns with an error. If the Monotonic Clock option is supported, the clock that is used to measure this time interval is the CLOCK MONOTONIC clock.

Parameters

list Array of asynchronous I/O operations.

nent Indicates the number of elements in the list array.

Specifies the time the subroutine has to complete the operation. timeout

Execution Envrionment

The aio suspend and aio suspend64 subroutines can be called from the process environment only.

Return Values

If the aio_suspend subroutine returns after one or more asynchronous I/O operations have completed, it returns a 0. Otherwise, it returns a -1 and sets the errno global variable to indicate the error.

The application can determine which asynchronous I/O completed by scanning the associated error and returning status using the aio error and aio return subroutines, respectively.

Error Codes

EAGAIN No asynchronous I/O indicated in the list referenced by *list* completed in the time interval indicated by

EINTR A signal interrupted the aio_suspend subroutine. Since each asynchronous I/O operation may possibly

provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited.

Related Information

"aio cancel or aio cancel64 Subroutine" on page 22, "aio error or aio error64 Subroutine" on page 25, "aio nwait Subroutine" on page 29, "aio nwait timeout Subroutine" on page 31, "aio read or aio read64 Subroutine" on page 33, "aio_return or aio_return64 Subroutine" on page 37, "aio_write or aio_write64 Subroutine" on page 43, and "lio listio or lio listio64 Subroutine" on page 611.

Legacy AIO aio_suspend Subroutine

Purpose

Suspends the calling process until one or more asynchronous I/O requests is completed.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
aio suspend( count, aiocbpa)
int count:
struct aiocb *aiocbpa[];
aio_suspend64( count, aiocbpa)
int count;
struct aiocb64 *aiocbpa[];
```

Description

The aio_suspend subroutine suspends the calling process until one or more of the count parameter asynchronous I/O requests are completed or a signal interrupts the subroutine. Specifically, the aio suspend subroutine handles requests associated with the aio control block (aiocb) structures pointed to by the aiocbpa parameter.

The aio_suspend64 subroutine is similar to the aio_suspend subroutine except that it takes an array of pointers to aiocb64 structures. This allows the aio_suspend64 subroutine to suspend on asynchronous I/O requests submitted by either the aio read64, aio write64, or the lio listio64 subroutines.

In the large file enabled programming environment, aio_suspend is redefined to be aio_suspend64.

The array of aiocb pointers may include null pointers, which will be ignored. If one of the I/O requests is already completed at the time of the aio_suspend call, the call immediately returns.

Note: The _AIO_AIX_SOURCE macro used in aio.h must be defined when using aio.h to compile an aio application with the Legacy AIO function definitions. The default compile using the aio.h file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define AIO AIX SOURCE
#include <sys/aio.h>
or, on the command line when compiling enter:
->xlc ... -D AIO AIX SOURCE ... legacy aio program.c
```

Parameters

count Specifies the number of entries in the aiocbpa array.

aiocbpa Points to the aiocb or aiocb64 structures associated with the asynchronous I/O operations.

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

```
int
                aio whence
off t
                aio offset
char
                *aio buf
                aio_nbytes
size_t
                aio_reqprio
int
struct event aio event
struct osigevent aio event
                aio flag
int
aiohandle t
                aio handle
```

Execution Envrionment

The aio_suspend and aio_suspend64 subroutines can be called from the process environment only.

Return Values

If one or more of the I/O requests completes, the **aio_suspend** subroutine returns the index into the aiocbpa array of one of the completed requests. The index of the first element in the *aiocbpa* array is 0. If more than one request has completed, the return value can be the index of any of the completed requests.

In the event of an error, the **aio_suspend** subroutine returns a value of -1 and sets the **errno** global variable to identify the error. Return codes can be set to the following **errno** values:

EINTR Indicates that a signal or event interrupted the **aio_suspend** subroutine call.

EINVAL Indicates that the aio_whence field does not have a valid value or that the resulting pointer is not valid.

Related Information

"aio_cancel or aio_cancel64 Subroutine" on page 22, "aio_error or aio_error64 Subroutine" on page 25, "aio_nwait Subroutine" on page 29, "aio_nwait_timeout Subroutine" on page 31, "aio_read or aio_read64 Subroutine" on page 33, "aio_return or aio_return64 Subroutine" on page 37, "aio_write or aio_write64 Subroutine," and "lio_listio or lio_listio64 Subroutine" on page 611.

The Asynchronous I/O Overview and the Communications I/O Subsystem: Programming Introduction in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

The Input and Output Handling Programmer's Overview in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs* describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

aio_write or aio_write64 Subroutine

The **aio_write** subroutine includes information for the POSIX AIO **aio_write** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **aio_write** subroutine.

POSIX AIO aio_write Subroutine

Purpose

Asynchronously writes to a file.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
int aio_write (aiocbp)
struct aiocb *aiocbp;
```

Description

The **aio_write** subroutine writes *aio_nbytes* to the file associated with *aio_fildes* from the buffer pointed to by *aio_buf*. The subroutine returns when the write request has been initiated or queued to the file or device.

The *aiocbp* parameter may be used as an argument to the **aio_error** and **aio_return** subroutines in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding.

The *aiocbp* parameter points to an **aiocb** structure. If the buffer pointed to by *aio_buf* or the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, the behavior is undefined.

If O_APPEND is not set for the <code>aio_fildes</code> file descriptor, the requested operation takes place at the absolute position in the file as given by <code>aio_offset</code>. This is done as if the <code>Iseek</code> subroutine were called immediately prior to the operation with an offset equal to <code>aio_offset</code> and a whence equal to <code>SEEK_SET</code>. If O_APPEND is set for the file descriptor, write operations append to the file in the same order as the calls were made. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The aio_lio_opcode field is ignored by the aio_write subroutine.

Since prioritized I/O is not supported at this time the *aio_reqprio* field of the **aiocb** structure is not presently used.

Simultaneous asynchronous operations using the same aiocbp produce undefined results.

If synchronized I/O is enabled on the file associated with *aio_fildes*, the behavior of this subroutine is according to the definitions of synchronized I/O data integrity completion, and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding, the result of that action is undefined.

For regular files, no data transfer occurs past the offset maximum established in the open file description associated with *aio_fildes*.

Parameters

aiocbp Points to the aiocb structure associated with the I/O operation.

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

int aio_fildes
off_t aio_offset
char *aio buf

size_t aio_nbytes int aio_reqprio struct sigevent aio_sigevent int aio_lio_opcode

Execution Environment

The aio_write and aio_write64 subroutines can be called from the process environment only.

Return Values

The **aio_write** subroutine returns a 0 to the calling process if the I/O operation is successfully queued. Otherwise, a -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

EAGAIN The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to **aio_write**, or asynchronously. If any of the conditions below are detected synchronously, the **aio_write** subroutine returns a -1 and sets the **errno** global variable to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation is set to the corresponding value.

EBADF The *aio_fildes* parameter is not a valid file descriptor open for writing.

EINVAL The file offset value implied by *aio_offset* is invalid, *aio_reaprio* is not a valid value, or *aio_nbytes* is

an invalid value.

If the **aio_write** subroutine successfully queues the I/O operation, the return status of the asynchronous operation is one of the values normally returned by the **write** subroutine call. If the operation is successfully queued but is subsequently canceled or encounters an error, the error status for the asynchronous operation contains one of the values normally set by the **write** subroutine call, or one of the following:

EBADF The *aio_fildes* parameter is not a valid file descriptor open for writing.

EINVAL The file offset value implied by *aio_offset* would be invalid.

ECANCELED The requested I/O was canceled before the I/O completed due to an aio_cancel request.

The following condition may be detected synchronously or asynchronously:

EFBIG The file is a regular file, *aio_nbytes* is greater than 0, and the starting offset in *aio_offset* is at or

beyond the offset maximum in the open file description associated with aio_fildes.

Related Information

"lio_listio or lio_listio64 Subroutine" on page 611, "aio_read or aio_read64 Subroutine" on page 33, "aio_nwait Subroutine" on page 29, "aio_nwait_timeout Subroutine" on page 31, "aio_return or aio_return64 Subroutine" on page 37, "close Subroutine" on page 141, "exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196, "exit, atexit, _exit, or _Exit Subroutine" on page 203, "fork, f_fork, or vfork Subroutine" on page 248, "aio_suspend or aio_suspend64 Subroutine" on page 40, and "lseek, llseek or lseek64 Subroutine" on page 646.

The read, ready, ready, or pread Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Legacy AIO aio_write Subroutine

Purpose

Writes to a file asynchronously.

Library

Standard C Library (libc.a)

Syntax

```
#include <aio.h>
int aio write( FileDescriptor, aiocbp)
int FileDescriptor;
struct aiocb *aiocbp;
int aio write64( FileDescriptor, aiocbp)
int FileDescriptor;
struct aiocb64 *aiocbp;
```

Description

The aio write subroutine writes asynchronously to a file. Specifically, the aio write subroutine writes to the file associated with the FileDescriptor parameter from a buffer. To handle this, the subroutine uses information from the aio control block (aiocb) structure, which is pointed to by the aiocbp parameter. This information includes the following fields:

```
aio buf
                                Indicates the buffer to use.
                                Indicates the number of bytes to write.
aio nbytes
```

The aio write64 subroutine is similar to the aio write subroutine except that it takes an aiocb64 reference parameter. This allows the aio_write64 subroutine to specify offsets in excess of OFF_MAX (2 gigbytes minus 1).

In the large file enabled programming environment, aio read is redefined to be aio read64.

When the write request has been queued, the aio_write subroutine updates the file pointer specified by the aio whence and aio offset fields in the aiocb structure as if the requested I/O completed. It then returns to the calling program. The aio whence and aio offset fields have the same meaning as the whence and offset parameters in the Iseek ("Iseek, Ilseek or Iseek64 Subroutine" on page 646) subroutine. The subroutine ignores them for file objects that are not capable of seeking.

If an error occurs during the call, the write request is not initiated or queued. To determine the status of a request, use the aio_error ("aio_error or aio_error64 Subroutine" on page 25) subroutine.

To have the calling process receive the **SIGIO** signal when the I/O operation completes, set the AIO_SIGNAL bit in the aio flag field in the aiocb structure.

Note: The event structure in the aiocb structure is currently not in use but is included for future compatibility.

Note: The _AIO_AIX_SOURCE macro used in aio.h must be defined when using aio.h to compile an aio application with the Legacy AIO function definitions. The default compile using the aio.h file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define AIO AIX SOURCE
#include <sys/aio.h>
```

```
or, on the command line when compiling enter:
->xlc ... -D AIO AIX SOURCE ... legacy aio program.c
```

Parameters

FileDescriptor Identifies the object to be written as returned from a call to open.

Points to the asynchronous I/O control block structure associated with the I/O operation. aiocbp

aiocb Structure

The aiocb structure is defined in the /usr/include/aio.h file and contains the following members:

aio whence off t aio offset char *aio buf aio nbytes size t aio_reqprio int aio event struct event struct osigevent aio_event aio flag aiohandle t aio handle

Execution Environment

The aio_write and aio_write64 subroutines can be called from the process environment only.

Return Values

When the write request gueues successfully, the aio write subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the errno global variable to identify the error.

Return codes can be set to the following **errno** values:

EAGAIN Indicates that the system resources required to queue the request are not available. Specifically, the

transmit queue may be full, or the maximum number of opens may have been reached.

EBADF Indicates that the FileDescriptor parameter is not valid.

EFAULT Indicates that the address specified by the aiocbp parameter is not valid.

EINVAL Indicates that the aio whence field does not have a valid value or that the resulting pointer is not

valid.

Note: Other error codes defined in the /usr/include/sys/errno.h file may be returned by the aio_error subroutine if an error during the I/O operation is encountered.

Related Information

"lio_listio or lio_listio64 Subroutine" on page 611, "aio_read or aio_read64 Subroutine" on page 33, "aio_nwait Subroutine" on page 29, "aio_nwait_timeout Subroutine" on page 31, "aio_return or aio_return64 Subroutine" on page 37, "close Subroutine" on page 141, "exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196, "exit, atexit, _exit, or _Exit Subroutine" on page 203, "fork, f fork, or vfork Subroutine" on page 248, "aio suspend or aio suspend64 Subroutine" on page 40, and "Iseek, Ilseek or Iseek64 Subroutine" on page 646.

The Asynchronous I/O Overview and the Communications I/O Subsystem: Programming Introduction in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

The Input and Output Handling Programmer's Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

alloc, dealloc, print, read_data, read_regs, symbol_addrs, write_data, and write_regs Subroutine

Purpose

Provide access to facilities needed by the pthread debug library and supplied by the debugger or application.

Library

pthread debug library (libpthdebug.a)

Syntax

```
#include <sys/pthdebug.h>
int alloc (user, len, bufp)
pthdb_user_t user;
size_t len;
void **bufp;
int dealloc (user, buf)
pthdb_user_t user;
void *buf;
int print (user, str)
pthdb user t user;
char *str;
int read data (user, buf, addr, size)
pthdb_user_t user;
void *buf;
pthdb_addr_t addr;
int size;
int read_regs (user, tid, flags, context)
pthdb_user_t user;
tid t tid;
unsigned long long flags;
struct context64 *context;
int symbol addrs (user, symbols[],count)
pthdb_user_t user;
pthdb_symbol_t symbols[];
int count;
int write_data (user, buf, addr, size)
pthdb_user_t user;
void *buf;
pthdb addr t addr;
int size;
int write regs (user, tid, flags, context)
pthdb_user_t user;
tid t tid;
unsigned long long flags;
struct context64 *context;
```

Description

int alloc()

Allocates *len* bytes of memory and returns the address. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is always required.

int dealloc()

Takes a buffer and frees it. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is always required.

int print()

Prints the character string to the debugger's stdout. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back is for debugging the library only. If you aren't debugging the pthread debug library, pass a NULL value for this call back.

int read_data()

Reads the requested number of bytes of data at the requested location from an active process or core file and returns the data through a buffer. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is always required.

int read_regs()

Reads the context information of a debuggee kernel thread from an active process or from a core file. The information should be formatted in **context64** form for both a 32-bit and a 64-bit process. If successful, 0 is returned; otherwise, a nonzero number is returned. This function is only required when using the **pthdb_pthread_context** and **pthdb_pthread_setcontext** subroutines.

int symbol_addrs()

Resolves the address of symbols in the debuggee. The pthread debug library calls this subroutine to get the address of known debug symbols. If the symbol has a name of NULL or "", set the address to 0LL instead of doing a lookup or returning an error. If successful, 0 is returned; otherwise, a nonzero number is returned. In introspective mode, when the **PTHDB_FLAG_SUSPEND** flag is set, the application can use the pthread debug library by passing NULL, or it can use one of its own.

int write_data()

Writes the requested number of bytes of data to the requested location. The **libpthdebug.a** library may use this to write data to the active process. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is required when the **PTHDB_FLAG_HOLD** flag is set and when using the **pthdb_pthread_setcontext** subroutine.

int write_regs()

Writes requested context information to specified debuggee's kernel thread id. If successful, 0 is returned; otherwise, a nonzero number is returned. This subroutine is only required when using the **pthdb_pthread_setcontext** subroutine.

Note: If the **write_data** and **write_regs** subroutines are NULL, the pthread debug library will not try to write data or regs. If the **pthdb_pthread_set_context** subroutine is called when the **write_data** and **write_regs** subroutines are NULL, **PTHDB_NOTSUP** is returned.

Parameters

user	User handle.
symbols	Array of symbols.
count	Number of symbols.
buf	Buffer.
addr	Address to be read from or wrote to.
size	Size of the buffer.
flags	Session flags, must accept PTHDB_FLAG_GPRS, PTHDB_FLAG_SPRS, PTHDB_FLAG_FPRS, and PTHDB_FLAG_REGS.
tid	Thread id.
flags	Flags that control which registers are read or wrote.
context	Context structure.
len	Length of buffer to be allocated or reallocated.
bufp	Pointer to buffer.
str	String to be printed.

Return Values

If successful, these subroutines return 0; otherwise they return a nonzero value.

Related Information

The "malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, or valloc Subroutine" on page 657.

arm_end Subroutine

Purpose

The **arm_end** subroutine is used to mark the end of an application. This subroutine call must always be called when a program that issued an **arm_init** ("arm_init Subroutine" on page 58) subroutine call terminates. In the PTX implementation of ARM, application data structures may persist after **arm_end** is issued.

Library

ARM Library (libarm.a).

Syntax

Description

By calling the **arm_end** subroutine, an application program signals to the ARM library that it has ceased issuing ARM subroutine calls for the application specified and that the library code can remove references to the application. As far as the calling program is concerned, all references to transactions defined for the named application can be removed as well.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that, in the PTX implementation of ARM, multiple processes can issue **arm_init** ("arm_init Subroutine" on page 58) subroutine calls for a given application with the effect that multiple simultaneous definitions of the application are effective. The ARM library code points all these definitions to a single application structure in the ARM private shared memory area. A use-count keeps track of the number of simultaneous definitions. Each time **arm_init** is issued for the application name, the counter is incremented and each time the **arm_end** subroutine call is issued for the associated <code>appl_id</code>, the counter is decremented. No call to **arm_end** is permitted to decrement the counter less than zero.

Only when the counter reaches zero is the application structure inactivated. As long as the counter is non-zero, transactions defined for the application remain active and new transactions can be defined for the application. It does not matter which process created the definition of the application.

This implementation was chosen because it makes perfect sense in a PTX environment. Any more restrictive implementation would have increased memory use significantly and would be useless for PTX monitoring purposes.

Parameters

appl_id

The identifier is returned by an earlier call to arm_init, "arm_init Subroutine" on page 58. The PTX implementation does not require that the arm init subroutine call was issued by the same program or process now issuing the arm end subroutine call. However, each time the arm end subroutine call is issued against an appl id, the use-count of the transaction structure is decremented. When the count reaches zero, the application structure and all associated transaction structures are marked as inactive. Subsequent arm init calls can reactivate the application structure but transaction structures formerly associated with the application are not automatically activated. Each transaction must be reactivated through the arm getid ("arm getid Subroutine" on page 53) subroutine call.

The appl_id is used to look for an application structure. If none is found, no action is taken and the function returns -1. If one is found, the use-count of the application structure is decremented. If that makes the counter zero, the use-counts of all associated transaction structures are set to zero. The total number of application structures that have been initialized for the calling process but not ended is decremented. If this count reaches zero, access to the shared memory from the process is released and the count of users of the shared memory area is decremented. If the count of users of the shared memory segment reaches zero, the shared memory segment is deleted.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

894, arm_init ("arm_init Subroutine" on page 58) subroutine, arm_getid ("arm_getid Subroutine" on page 53) subroutine.

arm_end Dual Call Subroutine

Purpose

The arm end subroutine is used to mark the end of an application. This subroutine call must always be called when a program that issued an arm_init ("arm_init Dual Call Subroutine" on page 60) subroutine call terminates. In the PTX implementation of ARM, application data structures may persist after arm_end is issued. This may not be the case for the *lower library* implementation.

Library

ARM Library (libarm.a).

Syntax

```
#include arm.h
arm ret stat t ARM API arm end( arm appl id t appl id,
                                                                      /* application id
                      flags, /* Reserved = 0
*data, /* Reserved = NUL
        arm flag t
       arm_data_t *data, /* Reserved = NULL
arm_data_sz_t data_size); /* Reserved = 0
                                                                                 */
```

Description

By calling the arm_end subroutine, an application program signals to the ARM library that it has ceased issuing ARM subroutine calls for the application specified and that the library code can remove references to the application. As far as the calling program is concerned, all references to transactions defined for the named application can be removed as well.

Before the PTX implementation code is executed, the *lower library* is called. If this call returns a value of zero, that return value is passed to the caller. If the value returned by the lower library is non-zero, the return value is the one generated by the PTX library code.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that, in the PTX implementation of ARM, multiple processes can issue arm init ("arm init Dual Call Subroutine" on page 60) subroutine calls for a given application with the effect that multiple simultaneous definitions of the application are effective. The ARM library code points all these definitions to a single application structure in the ARM private shared memory area. A use-count keeps track of the number of simultaneous definitions. Each time arm init is issued for the application name, the counter is incremented and each time the arm end subroutine call is issued for the associated appl id, the counter is decremented. No call to arm end is permitted to decrement the counter less than zero.

Only when the counter reaches zero is the application structure inactivated. As long as the counter is non-zero, transactions defined for the application remain active and new transactions can be defined for the application. It does not matter which process created the definition of the application.

This implementation was chosen because it makes perfect sense in a PTX environment. Any more restrictive implementation would have increased memory use significantly and would be useless for PTX monitoring purposes.

For the implementation of arm_end in the lower library, other restrictions may exist.

Parameters

appl_id

The identifier returned by an earlier call to arm_init ("arm_init Dual Call Subroutine" on page 60). The identifier is passed to the arm_end function of the lower library. If the lower library returns a zero, a zero is returned to the caller. After the invocation of the lower library, the PTX implementation attempts to translate the appl_id argument to its own identifier from the cross-reference table created by **arm init** ("arm init Dual Call Subroutine" on page 60). If one can be found, it is used for the PTX implementation; if no cross reference is found, the appl_id is used as passed in. The PTX implementation does not require that the arm_init subroutine call was issued by the same program or process now issuing the arm_end subroutine call. However, each time the arm end subroutine call is issued against an appl id, the use-count of the transaction structure is decremented. When the count reaches zero, the application structure and all associated transaction structures are marked as inactive. Subsequent arm init calls can reactivate the application structure but transaction structures formerly associated with the application are not

automatically activated. Each transaction must be reactivated through the **arm_getid** ("arm_getid Dual Call Subroutine" on page 55) subroutine call.

In the PTX implementation, the <code>appl_id</code> (as retrieved from the cross-reference table) is used to look for an application structure. If none is found, no action is taken and the PTX function is considered to have failed. If one is found, the use-count of the application structure is decremented. If that makes the counter zero, the use-counts of all associated transaction structures are set to zero. The total number of application structures that have been initialized for the calling process but not ended is decremented. If this count reaches zero, access to the shared memory from the process is released and the count of users of the shared memory area is decremented. If the count of users of the shared memory segment reaches zero, the shared memory segment is deleted.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned. If the call to the *lower library* was successful, a zero is returned. If the subroutine call to the *lower library* failed but the PTX implementation didn't fail, a zero is returned. If both implementations failed, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

- · "arm_init Dual Call Subroutine" on page 60
- "arm_getid Dual Call Subroutine" on page 55

arm_getid Subroutine

Purpose

The **arm_getid** subroutine is used to register a transaction as belonging to an application and assign a unique identifier to the application/transaction pair. In the PTX implementation of ARM, multiple instances of a transaction within one application can't be defined. A transaction must be registered before any ARM measurements can begin.

Library

ARM Library (libarm.a).

Syntax

```
arm_ptr_t *tran_detail, /* transaction additional info
arm_flag_t flags, /* Reserved = 0
arm_data_t *data, /* Reserved = NULL
arm_data_sz_t data_size); /* Reserved = 0
```

Description

Each transaction needs to be defined by a unique name within an application. Transactions can be defined so they best fit the application environment. For example, if a given environment has thousands of unique transactions, it may be feasible to define groups of similar transactions to prevent data overload. In other situations, you may want to use generated transaction names that reflect what data a transaction carries along with the transaction type. For example, the type of SQL guery could be analyzed to group customer query transactions according to complexity, such as customer simple, customer, customer complex. Whichever method is used to name transactions, in the PTX implementation of the ARM API, measurements are always collected for each unique combination of:

- 1. Hostname of the machine where the instrumented application executes.
- 2. Unique application name.
- 3. Unique transaction name.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the use-count for a transaction structure is either one or zero. This ensures that as long as the application structure is active, so are all transactions for which an arm_getid call was issued after the application was activated by an arm_init ("arm_init Subroutine" on page 58) call. The transaction use-count is reset to zero by the arm_end ("arm_end Subroutine" on page 50) call if this call causes the application use-count to go to zero.

Note that the implementation of arm_getid doesn't allow unique instances of a transaction to be defined. The tran_id associated with a transaction is stored in the ARM shared memory area and will remain constant throughout the life of the shared memory area. Consequently, subsequent executions of a program that defines one or more transactions under a given application will usually have the same ID returned for the transactions each time. The same is true when different programs define the same transaction within an application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for transaction definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate transaction names to pass on the arm getid subroutine call.

Parameters

appl_id

The identifier returned by an earlier call to arm init ("arm init Subroutine" on page 58). The PTX implementation does not require that the arm init subroutine call was issued by the same program or process now issuing the arm getid subroutine call. However, the number of issued arm init subroutine calls for the application name must exceed the number of issued arm end subroutine calls for this appl_id.

The appl id is used to look for an application structure. If one is not found or if the use-count of the one found is zero, no action is taken and the function returns -1.

tran name

A unique transaction name. The name only needs to be unique within the appl_id. The maximum length is 128 characters including the terminating zero. The argument is converted to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This

key is used to look for a transaction structure (that belongs to the application identified in the first argument) in the library's private shared memory area. If a transaction structure is found, its use-count is set to one and the transaction ID stored in the structure is returned to the caller. If the structure is not found, one is created and assigned the next free transaction ID, given a use-count of one and added to the application's linked list of transactions. The new assigned transaction ID is returned to the caller.

Up-to 64 bytes, including the terminating zero, of the tran_name parameter is saved as the description of the SpmiCx context that represents the transaction in the Spmi hierarchy. The key is used as the short name of the context.

tran detail

Can be passed in as NULL or some means of specifying a unique instance of the transaction. In the PTX implementation of the ARM API, this parameter is ignored. Consequently, it is not possible to define unique instances of a transaction. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes in length, including the terminating zero.

For the implementation to take this argument in use, another context level would have to be defined between the application context and the transaction context. This was deemed excessive.

flags, data, data size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns an tran_id application identifier. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the arm start ("arm start Subroutine" on page 61) subroutine, which will cause arm start to function as a no-operation.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

arm init ("arm init Subroutine" on page 58) subroutine, arm end ("arm end Subroutine" on page 50) subroutine.

arm_getid Dual Call Subroutine

Purpose

The arm_getid subroutine is used to register a transaction as belonging to an application and assign a unique identifier to the application/transaction pair. In the PTX implementation of ARM, multiple instances of a transaction within one application can't be defined. The lower library implementation of this subroutine may provide support for instances of transactions. A transaction must be registered before any ARM measurements can begin.

Library

ARM Library (libarm.a).

Syntax

```
#include arm.h
                                                                                                              /* application handle
arm tran id t arm getid(
                                                   arm appl id t appl id,
*/
            arm_ptr_t *tran_name, /* transaction name
arm_ptr_t *tran_detail, /* transaction additional info
arm_flag_t flags, /* Reserved = 0
arm_data_t *data, /* Reserved = NULL
arm_data_sz_t data_size); /* Reserved = 0
                                                                                                                                */
                                                                                                                                 */
                                                                                                                                 */
```

Description

Each transaction needs to be defined by a unique name within an application. Transactions can be defined so they best fit the application environment. For example, if a given environment has thousands of unique transactions, it may be feasible to define groups of similar transactions to prevent data overload. In other situations, you may want to use generated transaction names that reflect what data a transaction carries along with the transaction type. For example, the type of SQL query could be analyzed to group customer query transactions according to complexity, such as customer simple, customer, customer complex. Whichever method is used to name transactions, in the PTX implementation of the ARM API, measurements are always collected for each unique combination of:

- 1. Hostname of the machine where the instrumented application executes.
- 2. Unique application name.
- Unique transaction name.

Before the PTX implementation code is executed, the *lower library* is called. If this call returns a value greater than zero, that return value is passed to the caller as the transaction ID. If the returned value from the lower library is zero or negative, the return value is the one generated by the PTX library code.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the use-count for a transaction structure is either one or zero. This ensures that as long as the application structure is active, so are all transactions for which an arm_getid call was issued after the application was activated by an arm init ("arm init Dual Call Subroutine" on page 60) call. The transaction use-count is reset to zero by the arm_end ("arm_end Dual Call Subroutine" on page 51) call if this call causes the application use-count to go to zero.

Note that the implementation of arm getid doesn't allow unique instances of a transaction to be defined. The tran id associated with a transaction is stored in the ARM shared memory area and will remain constant throughout the life of the shared memory area. Consequently, subsequent executions of a program that defines one or more transactions under a given application will usually have the same ID returned for the transactions each time. The same is true when different programs define the same transaction within an application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for transaction definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate transaction names to pass on the arm_getid subroutine call.

Regardless of the implementation restrictions of the PTX library, the lower library may or may not have its own implementation restrictions.

Parameters

appl_id

The identifier returned by an earlier call to arm_init ("arm_init Dual Call Subroutine" on page 60). The identifier is passed to the arm getid function of the lower library. If the lower library returns an identifier greater than zero, that identifier is the one that'll eventually be returned to the caller. After the invocation of the lower library, the PTX implementation attempts to translate the appl_id argument to its own identifier by consulting the cross-reference table created by arm_init. If one can be found, it is used for the PTX implementation; if no cross reference is found, the appl id is used as passed in. The PTX implementation does not require that the arm_init subroutine call was issued by the same program or process now issuing the arm **getid** subroutine call. However, the number of issued arm_init subroutine calls for the application name must exceed the number of issued **arm_end** subroutine calls for this *appl_id*.

In the PTX implementation, the appl id (as retrieved from the cross-reference table) is used to look for an application structure. If one is not found or if the use-count of the one found is zero, the PTX implementation is considered to have failed and no action is taken by the PTX library.

tran_name

A unique transaction name. The name only needs to be unique within the appl id. The maximum length is 128 characters including the terminating zero. In the PTX implementation, the argument is converted to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This key is used to look for a transaction structure (that belongs to the application identified in the first argument) in the library's private shared memory area. If a transaction structure is found, its use-count is set to one and the transaction ID stored in the structure is saved. If the structure is not found, one is created and assigned the next free transaction ID, given a use-count of one and added to the application's linked list of transactions. The new assigned transaction ID is saved. If the call to the lower library was successful, a cross-reference is created from the lower library's transaction ID to the PTX library's transaction ID for use by arm start ("arm start Dual Call Subroutine" on page 63).

Up-to 64 bytes, including the terminating zero, of the tran name parameter is saved as the description of the SpmiCx context that represents the transaction in the Spmi hierarchy. The key is used as the short name of the context.

tran_detail

Can be passed in as NULL or some means of specifying a unique instance of the transaction. In the PTX implementation of the ARM API, this parameter is ignored. Consequently, it is not possible to define unique instances of a transaction. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes in length, including the terminating zero.

For the implementation to take this argument in use, another context level would have to be defined between the application context and the transaction context. This was deemed excessive.

For the lower library implementation of this subroutine call, the tran_detail argument may have significance. If so, it's transparent to the PTX implementation.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation. In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns an tran id application identifier. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the arm start ("arm start Dual Call Subroutine" on page 63) subroutine, which will cause arm start to function as a no-operation.

If the call to the lower library was successful, the tran id transaction identifier returned is the one assigned by the lower library. If the subroutine call to the lower library failed but the PTX implementation didn't fail, the tran id returned is the one assigned by the PTX library. If both implementations fail, a value less than zero is returned. In compliance with the ARM API specification, an error return value can be passed to the arm_start ("arm_start Dual Call Subroutine" on page 63) subroutine, which will cause arm_start to function as a no-operation.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

arm_init ("arm_init Dual Call Subroutine" on page 60) subroutine, arm_end ("arm_end Dual Call Subroutine" on page 51) subroutine.

arm init Subroutine

Purpose

The arm init subroutine is used to define an application or a unique instance of an application to the ARM library. In the PTX implementation of ARM, instances of applications can't be defined. An application must be defined before any other ARM subroutine is issued.

Library

ARM Library (libarm.a).

Syntax

```
#include arm.h
arm_appl_id_t arm_init( arm_ptr_t *appname,
                                                                    /* application name
                         *appl user id, /* Name of the application user */
         arm ptr t
        arm_flag_t flags, /* Reserved = 0
arm_data_t *data, /* Reserved = NULL
arm_data_sz_t data_size); /* Reserved = 0
                                                                                         */
                                                                                         */
```

Description

Each application needs to be defined by a unique name. An application can be defined as loosely or as rigidly as required. It may be defined as a single execution of one program, multiple (possibly simultaneous) executions of one program, or multiple executions of multiple programs that together constitute an application. Any one user of ARM may define the application so it best fits the measurement granularity desired. Measurements are always collected for each unique combination of:

- 1. Hostname of the machine where the instrumented application executes.
- 2. Unique application name.
- Unique transaction name.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the implementation of arm init doesn't allow unique instances of an application to be defined. The appl id associated with an application is stored in the ARM shared memory area and will remain

constant throughout the life of the shared memory area. Consequently, subsequent executions of a program that defines one or more applications will usually have the same ID returned for the application each time. The same is true when different programs define the same application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for application definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate application names to pass on the **arm init** subroutine call.

Parameters

appname

A unique application name. The maximum length is 128 characters including the terminating zero. The argument is converted to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This key is used to look for an application structure in the library's private shared memory area. If a structure is found, its use-count is incremented and the application ID stored in the structure is returned to the caller. If the structure is not found, one is created, assigned the next free application ID and given a use-count of one. The new assigned application ID is returned to the caller.

Up-to 64 bytes, including the terminating zero, of the *appname* parameter is saved as the description of the **SpmiCx context** that represents the application in the Spmi hierarchy. The key is used as the short name of the context.

appl user id

Can be passed in as NULL or some means of specifying a user ID for the application. This allows the calling program to define unique instances of an application. In the PTX implementation of the ARM API, this parameter is ignored. Consequently, it is not possible to define unique instances of an application. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes in length, including the terminating zero.

For the implementation to take this argument in use, another context level would have to be defined between the application context and the transaction context. This was deemed excessive.

flags, data, data size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns an **appl_id** application identifier. If the subroutine fails, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

arm_init Dual Call Subroutine

Purpose

The **arm_init** subroutine is used to define an application or a unique instance of an application to the ARM library. While, in the PTX implementation of ARM, instances of applications can't be defined, the ARM implementation in the *lower library* may permit this. An application must be defined before any other ARM subroutine is issued.

Library

ARM Library (libarm.a).

Syntax

```
#include arm.h
arm_appl_id_t arm_init( arm_ptr_t *appname, /* application name
*/
    arm_ptr_t *appl_user_id, /* Name of the application user */
    arm_flag_t flags, /* Reserved = 0 */
    arm_data_t *data, /* Reserved = NULL */
    arm data sz t data size); /* Reserved = 0 */
```

Description

Each application needs to be defined by a unique name. An application can be defined as loosely or as rigidly as required. It may be defined as a single execution of one program, multiple (possibly simultaneous) executions of one program, or multiple executions of multiple programs that together constitute an application. Any one user of ARM may define the application so it best fits the measurement granularity desired. For the PTX implementation, measurements are always collected for each unique combination of:

- 1. Hostname of the machine where the instrumented application executes.
- 2. Unique application name.
- 3. Unique transaction name.

Before the PTX implementation code is executed, the *lower library* is called. If this call returns a value greater than zero, that return value is passed to the caller as the application ID. If the returned value from the *lower library* is zero or negative, the return value is the one generated by the PTX library code.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the implementation of **arm_init** doesn't allow unique instances of an application to be defined. The **appl_id** associated with an application is stored in the ARM shared memory area and will remain constant throughout the life of the shared memory area. Consequently, subsequent executions of a program that defines one or more applications will usually have the same ID returned for the application each time. The same is true when different programs define the same application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for application definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate application names to pass on the **arm_init** subroutine call.

Regardless of the implementation restrictions of the PTX library, the *lower library* may or may not have its own implementation restrictions.

Parameters

appname

A unique application name. The maximum length is 128 characters including the terminating zero. The PTX library code converts this value to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This key is used to look for an application structure in the library's private shared memory area. If a structure is found, its use-count is incremented and the application ID stored in the structure is saved. If the structure is not found, one is created, assigned the next free application ID and given a use-count of one. The new assigned application ID is saved. If the call to the lower library was successful, a cross-reference is created from the lower library's application ID to the PTX library's application ID for use by arm getid ("arm getid Dual Call Subroutine" on page 55) and arm_end ("arm_end Dual Call Subroutine" on page 51).

Up-to 64 bytes, including the terminating zero, of the appname parameter is saved as the description of the SpmiCx context that represents the application in the Spmi hierarchy. The key is used as the short name of the context.

appl_user_id

Can be passed in as NULL or some means of specifying a user ID for the application. This allows the calling program to define unique instances of an application. In the PTX implementation of the ARM API, this parameter is ignored. Consequently, it is not possible to define unique instances of an application. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes in length, including the terminating zero.

For the PTX implementation to take this argument in use, another context level would have to be defined between the application context and the transaction context. This was deemed excessive.

For the lower library implementation of this subroutine call, the appl_user_id argument may have significance. If so, it's transparent to the PTX implementation.

flags, data, data size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If the call to the *lower library* was successful, the subroutine returns an appl id application identifier as returned from the *lower library*. If the subroutine call to the *lower library* fails but the PTX implementation doesn't fail, the appl id returned is the one assigned by the PTX library. If both implementations fail, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

arm_start Subroutine

Purpose

The arm start subroutine is used to mark the beginning of the execution of a transaction. Measurement of the transaction response time starts at the execution of this subroutine.

Library

ARM Library (libarm.a).

Syntax

```
#include arm.h
arm start handle t arm start( arm tran id t tran id, /* transaction name identifier

/* Reserved = 0
/* Reserved
/* Reserved
                     flags,
       arm flag t
       arm_data_t *data, /* Reserved = NULL
arm_data_sz_t data_size); /* Reserved = 0
                                                                                */
```

Description

Each arm_start subroutine call marks the beginning of another instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held until the execution of a matching arm stop ("arm stop Subroutine" on page 65) subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:

- 1. Hostname of the machine where the instrumented application executes.
- 2. Unique application name.
- 3. Unique transaction name.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Parameters

tran id

The identifier is returned by an earlier call to arm getid, "arm getid Subroutine" on page 53. The PTX implementation does not require that the arm getid subroutine call was issued by the same program or process now issuing the arm start subroutine call. However, the transaction's application structure must be active, which means that the number of issued arm init subroutine calls for the application name must exceed the number of issued arm end subroutine calls for the application's appl id. If an application was inactivated by issuing a sufficient number of arm end calls, all transactions defined for that application will have their use count set to zero. The count remains zero (and the transaction inactive) until a new arm getid subroutine is issued for the transaction.

The tran_id argument is used to look for a transaction structure. If one is not found or if the use-count of the one found is zero, no action is taken and the function returns -1. If one is found, a transaction instance structure (called a slot structure) is allocated, assigned the next free instance ID, and updated with the start time of the transaction instance. The assigned instance ID is returned to the caller.

In compliance with the ARM API specifications, if the tran_id passed is one returned from a previous arm_getid subroutine call that failed, the arm_start subroutine call functions as a no-operation function. It will return a NULL start_handle, which can be passed to subsequent arm_update ("arm_update Subroutine" on page 68) and arm_stop ("arm_stop Subroutine" on page 65) subroutine calls with the effect that those calls are no-operation functions.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns a start_handle, which uniquely defines this transaction execution instance. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the arm update ("arm update Subroutine" on page 68) and arm stop ("arm stop Subroutine" on page 65) subroutines, which will cause those subroutines to operate as no-operation functions.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

arm_init ("arm_init Subroutine" on page 58) subroutine, arm_getid ("arm_getid Subroutine" on page 53) subroutine, **arm end** ("arm end Subroutine" on page 50) subroutine.

arm start Dual Call Subroutine

Purpose

The arm start subroutine is used to mark the beginning of the execution of a transaction. Measurement of the transaction response time starts at the execution of this subroutine.

Library

ARM Library (libarm.a).

Syntax

```
#include arm.h
arm start handle t arm start( arm tran id t tran id, /* transaction name identifier
                               /* Reserved = 0
      arm flag t
                  flags,
                          /* Reserved = NULL
      arm_data_t *data,
      arm data sz t data size); /* Reserved = 0
```

Description

Each arm_start subroutine call marks the beginning of another instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held until the execution of a matching arm stop ("arm stop Dual Call Subroutine" on page 66) subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:

- 1. Hostname of the machine where the instrumented application executes.
- 2. Unique application name.
- 3. Unique transaction name.

Before the PTX implementation code is executed, the lower library is called. If this call returns a value greater than zero, that return value is passed to the caller as the start handle. If the value returned by the lower library is zero or negative, the return value is the one generated by the PTX library code.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Parameters

tran id

The identifier is returned by an earlier call to arm_getid, "arm_getid Dual Call Subroutine" on page 55. The identifier is passed to the arm start function of the lower library. If the lower library returns an identifier greater than zero, that identifier is the one that'll eventually be returned to the caller. After the invocation of the lower library, the PTX implementation attempts to translate the tran id argument to its own identifier from the cross-reference table created by arm getid. If one can be found, it is used for the PTX implementation; if no cross reference is found, the tran idis used as passed in. The PTX implementation does not require that the arm getid subroutine call was issued by the same program or process now issuing the arm_start subroutine call. However, the transaction's application structure must be active, which means that the number of issued arm_init subroutine calls for the application name must exceed the number of issued arm_end subroutine calls for the application's appl_id. If an application was inactivated by issuing a sufficient number of arm end calls, all transactions defined for that application will have their use_count set to zero. The count remains zero (and the transaction inactive) until a new **arm getid** subroutine is issued for the transaction.

In the PTX implementation, the tran_id (as retrieved from the cross-reference table) is used to look for a transaction structure. If one is not found or if the use-count of the one found is zero, the PTX implementation is considered to have failed and no action is taken by the PTX library. If one is found, a transaction instance structure (called a slot structure) is allocated, assigned the next free instance ID, and updated with the start time of the transaction instance. The assigned instance ID is saved as the **start_handle**. If the call to the *lower library* was successful, a cross-reference is created from the lower library's start handle to the PTX library's start handle for use by arm_update ("arm_update Dual Call Subroutine" on page 69) and arm_stop ("arm_stop Dual Call Subroutine" on page 66).

In compliance with the ARM API specifications, if the tran id passed is one returned from a previous arm_getid subroutine call that failed, the arm_start subroutine call functions as a no-operation function. It will return a NULL start handle, which can be passed to subsequent arm update ("arm update Dual Call Subroutine" on page 69) and arm stop ("arm stop Dual Call Subroutine" on page 66) subroutine calls with the effect that those calls are no-operation functions.

flags, data, data size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation. In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns a start_handle, which uniquely defines this transaction execution instance. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the arm_update ("arm_update Dual Call Subroutine" on page 69) and arm_stop ("arm_stop Dual Call Subroutine" on page 66) subroutines, which will cause those subroutines to operate as no-operation functions.

If the call to the *lower library* was successful, the **start_handle** instance ID returned is the one assigned by the lower library. If the subroutine call to the lower library failed but the PTX implementation didn't fail, the start handle returned is the one assigned by the PTX library. If both implementations fail, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

arm_init ("arm_init Dual Call Subroutine" on page 60) subroutine, arm_getid ("arm_getid Dual Call Subroutine" on page 55) subroutine, arm_end ("arm_end Dual Call Subroutine" on page 51) subroutine.

arm_stop Subroutine

Purpose

The **arm_stop** subroutine is used to mark the end of the execution of a transaction. Measurement of the transaction response time completes at the execution of this subroutine.

Library

ARM Library (libarm.a).

Syntax

Description

Each **arm_stop** subroutine call marks the end of an instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held from the execution of the **arm_start** ("arm_start Subroutine" on page 61) subroutine call and until the execution of a matching **arm_stop** subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:

- 1. Hostname of the machine where the instrumented application executes.
- 2. Unique application name.
- 3. Unique transaction name.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Parameters

arm handle

The identifier is returned by an earlier call to **arm_start**, "arm_start Subroutine" on page 61. The *arm_handle* argument is used to look for a *slot structure* created by the **arm_start** ("arm_start Subroutine" on page 61) call, which returned this *arm_handle*. If one is not found, no action is taken and the function returns -1. If one is found, a *post structure* is allocated and added to the linked list of post structures used to pass data to the **SpmiArmd** daemon. The post structure is updated with the start time from the slot structure, the path to the transaction context, and the stop time of the transaction instance.

In compliance with the ARM API specifications, if the start handle passed is one returned from a previous arm start subroutine call that failed, or from an arm start subroutine operating as a no-operation function, the arm_stop subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

comp status

User supplied transaction completion code. The following codes are defined:

- ARM GOOD successful completion. Response time is calculated. The response time is calculated as a fixed point value in milliseconds and saved in the metric resptime. In addition, the weighted average response time is calculated as a floating point value using a variable weight that defaults to 75%. The average response time is calculated as weight percent of the previous value of the average plus (100 - weight) percent of the latest response time observation. The value of weight can be changed from the **SpmiArmd** daemon's configuration file /etc/perf/SpmiArmd.cf. In addition, the maximum and minimum response time for this transaction is updated, if required. Finally the count of successful transaction executions is incremented.
- ARM ABORT transaction aborted. The aborted counter is incremented. No other updates occur.
- ARM FAILED transaction failed. The failed counter is incremented. No other updates occur.

flags, data, data size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

arm_init ("arm_init Subroutine" on page 58) subroutine, arm_getid ("arm_getid Subroutine" on page 53) subroutine, arm_start ("arm_start Subroutine" on page 61) subroutine, arm_end ("arm_end Subroutine" on page 50) subroutine.

arm stop Dual Call Subroutine

Purpose

The arm stop subroutine is used to mark the end of the execution of a transaction. Measurement of the transaction response time completes at the execution of this subroutine.

Library

ARM Library (libarm.a).

Syntax

Description

Each **arm_stop** subroutine call marks the end of an instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held from the execution of the **arm_start** ("arm_start Dual Call Subroutine" on page 63) subroutine call and until the execution of a matching **arm_stop** subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:

- 1. Hostname of the machine where the instrumented application executes.
- 2. Unique application name.
- 3. Unique transaction name.

Before the PTX implementation code is executed, the *lower library* is called. If this call returns a value of zero, that return value is passed to the caller. If the value returned by the *lower library* is non-zero, the return value is the one generated by the PTX library code.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Parameters

arm handle

The identifier is returned by an earlier call to **arm_start**, "arm_start Dual Call Subroutine" on page 63. The identifier is passed to the **arm_stop** function of the *lower library*. If the *lower library* returns a zero return code, that return code is returned to the caller. After the invocation of the *lower library*, the PTX implementation attempts to translate the *arm_handle* argument to its own identifier from the cross-reference table created by **arm_start**. If one can be found, it is used for the PTX implementation; if no cross reference is found, the *arm_handle* is used as passed in. The PTX implementation uses the *start_handle* argument to look for the *slot structure* created by the **arm_start** subroutine call. If one is found, a *post structure* is allocated and added to the linked list of post structures used to pass data to the **SpmiArmd** daemon. The post structure is updated with the start time from the slot structure, the path to the transaction context, and the stop time of the transaction instance. If no *slot structure* was found, the PTX implementation is considered to have failed.

In compliance with the ARM API specifications, if the *start_handle* passed is one returned from a previous **arm_start** subroutine call that failed, or from an **arm_start** subroutine operating as a no-operation function, the **arm_stop** subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

comp_status

User supplied transaction completion code. The following codes are defined:

• **ARM_GOOD** - successful completion. Response time is calculated. The response time is calculated as a fixed point value in milliseconds and saved in the metric **resptime**. In addition, the weighted average response time (in **respavg**) is calculated as a floating point value using a variable *weight*, that defaults to 75%. The average response time is calculated as *weight*

percent of the previous value of the average plus (100 - weight) percent of the latest response time observation. The value of weight can be changed from the SpmiArmd daemon's configuration file /etc/perf/SpmiArmd.cf. In addition, the maximum and minimum response time for this transaction is updated, if required. Finally the count of successful transaction executions is incremented.

- ARM_ABORT transaction aborted. The aborted counter is incremented. No other updates
- ARM_FAILED transaction failed. The failed counter is incremented. No other updates occur.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation. In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned. If the call to the lower library was successful, a zero is returned. If the subroutine call to the lower library failed but the PTX implementation didn't fail, a zero is returned. If both implementations failed, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

arm_init ("arm_init Dual Call Subroutine" on page 60) subroutine, arm_getid ("arm_getid Dual Call Subroutine" on page 55) subroutine, arm_start ("arm_start Dual Call Subroutine" on page 63) subroutine, arm end ("arm end Dual Call Subroutine" on page 51) subroutine.

arm_update Subroutine

Purpose

The arm_update subroutine is used to collect information about a transaction's progress. It is a no-operation subroutine in the PTX implementation.

Library

ARM Library (libarm.a).

Syntax

```
#include arm.h
arm ret stat t arm update( arm start handle t arm handle, /* unique transaction handle
                                 flags,
                                                /* Reserved = 0
        arm flag t
        arm_ftag_t ftags, /* Reserved = 0
arm_data_t *data, /* Reserved = Nt
arm_data_sz_t data_size); /* Reserved = 0
                                             /* Reserved = NULL
```

Description

The **arm_update** subroutine is implemented as a no-operation in the PTX version of the ARM API. It is intended to be used for providing status information for a long-running transaction. Because there's no feasible way to display such information in current PTX monitors, the subroutine is a NULL function.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product. It is implemented as a NULL subroutine call.

Parameters

start_handle

The identifier is returned by an earlier call to **arm_start**, "arm_start Subroutine" on page 61. The *start_handle* argument is used to look for the *slot structure* created by the **arm_start** subroutine call. If one is not found, no action is taken and the function returns -1. Otherwise a zero is returned.

In compliance with the ARM API specifications, if the *start_handle* passed is one returned from a previous **arm_start** subroutine call that failed, or from an **arm_start** subroutine operating as a no-operation function, the **arm_update** subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

arm_init ("arm_init Subroutine" on page 58) subroutine, arm_getid ("arm_getid Subroutine" on page 53) subroutine, arm_start ("arm_start Subroutine" on page 61) subroutine, arm_stop ("arm_stop Subroutine" on page 65) subroutine, arm_end ("arm_end Subroutine" on page 50) subroutine.

arm_update Dual Call Subroutine

Purpose

The **arm_update** subroutine is used to collect information about a transaction's progress. It is a no-operation subroutine in the PTX implementation but may be fully implemented by the *lower library*.

Library

ARM Library (libarm.a).

Syntax

Description

The **arm_update** subroutine is implemented as a no-operation in the PTX version of the ARM API. It is intended to be used for providing status information for a long-running transaction. Because there's no feasible way to display such information in current PTX monitors, the subroutine is a NULL function.

The lower library implementation of the arm_update subroutine is always invoked.

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product. It is implemented as a NULL subroutine call.

Parameters

start_handle

The identifier is returned by an earlier call to **arm_start**, "arm_start Dual Call Subroutine" on page 63. The identifier is passed to the **arm_update** function of the *lower library*. If the *lower library* returns a zero return code., that return code is returned to the caller. After the invocation of the *lower library*, the PTX implementation attempts to translate the *arm_handle*argument to its own identifier from the cross-reference table created by **arm_start**. If one can be found, it is used for the PTX implementation; if no cross reference is found, the *arm_handle* is used as passed in. The PTX implementation uses the *start_handle* argument to look for the *slot structure* created by the **arm_start** subroutine call. If one is found the PTX implementation is considered to have succeeded, otherwise it is considered to have failed.

In compliance with the ARM API specifications, if the *start_handle* passed is one returned from a previous **arm_start** subroutine call that failed, or from an **arm_start** subroutine operating as a no-operation function, the **arm_update** subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

flags, data, data size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation. In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned. If the call to the *lower library* was successful, a zero is returned. If the subroutine call to the *lower library* failed but the PTX implementation didn't fail, a zero is returned. If both implementations failed, a value less than zero is returned.

Error Codes

No error codes are defined by the PTX implementation of the ARM API.

Files

/usr/include/arm.h

Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

Related Information

arm_init ("arm_init Dual Call Subroutine" on page 60) subroutine, arm_getid ("arm_getid Dual Call Subroutine" on page 55) subroutine, arm_start ("arm_start Dual Call Subroutine" on page 63) subroutine, arm stop ("arm stop Dual Call Subroutine" on page 66) subroutine, arm end ("arm end Dual Call Subroutine" on page 51) subroutine.

asinh, asinhf, or asinhl Subroutine

Purpose

Computes the inverse hyperbolic sine.

Syntax

```
#include <math.h>
float asinhf (x)
float x;
long double asinhl (x)
long double x;
double asinh (x)
double x;
```

Description

The **asinhf**, asinhl, and asinh subroutines compute the inverse hyperbolic sine of thex parameter.

An application wishing to check for error situations should set errno to zero and call fetestexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if the errno global variable is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the asinhf, asinhl, and asinh subroutines return the inverse hyperbolic sine of the given argument.

If x is NaN, a NaN is returned.

If x is 0, or $\pm \ln x$, x is returned.

If x is subnormal, a range error may occur and x will be returned.

Related Information

math.h in AIX 5L Version 5.2 Files Reference.

asinf, asinl, or asin Subroutine

Purpose

Computes the arc sine.

Syntax

```
#include <math.h>
float asinf (x)
float x;
long double asinl (x)
long double x;
double as in (x)
double x;
```

Description

The asinf, asinf, and asin subroutines compute the principal value of the arc sine of the x parameter. The value of x should be in the range [-1,1].

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. On return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the asinf, asinf, and asin subroutines return the arc sine of x, in the range [-pi /2, pi/2] radians.

For finite values of x not in the range [-1,1], a domain error occurs, and a NaN is returned.

If x is NaN, a NaN is returned.

If x is 0, x is returned.

If x is $\pm \ln n$, a domain error occurs, and a NaN is returned.

If x is subnormal, a range error may occur and x is returned.

Related Information

The "asinh, asinhf, or asinhl Subroutine" on page 71.

math.h in AIX 5L Version 5.2 Files Reference.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

assert Macro

Purpose

Verifies a program assertion.

Library

Standard C Library (libc.a)

Syntax

```
#include <assert.h>
void assert ( Expression)
int Expression;
```

Description

The assert macro puts error messages into a program. If the specified expression is false, the assert macro writes the following message to standard error and stops the program:

```
Assertion failed: Expression, file FileName, line LineNumber
```

In the error message, the FileName value is the name of the source file and the LineNumber value is the source line number of the assert statement.

Parameters

Expression

Specifies an expression that can be evaluated as true or false. This expression is evaluated in the same manner as the C language IF statement.

Related Information

The **abort** ("abort Subroutine" on page 3) subroutine.

The cpp command.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

atan2f, atan2l, or atan2 Subroutine

Purpose

Computes the arc tangent.

Syntax

```
#include <math.h>
float atan2f (y, x)
float y, float x;
long double atan21 (y, x)
long double y, long double x;
double atan2 (y, x)
double y, x;
```

Description

The atan2f, atan2l, and atan2 subroutines compute the principal value of the arc tangent of y/x, using the signs of both parameters to determine the quadrant of the return value.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE ALL EXCEPT) before calling these functions. On return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to compute. Specifies the value to compute. Х

Return Values

Upon successful completion, the atan2f, atan2l, and atan2 subroutines return the arc tangent of y/x in the range [-pi, pi] radians.

If y is 0 and x is < 0, $\pm pi$ is returned.

If y is 0 and x is > 0, 0 is returned.

If y is < 0 and x is 0, -pi/2 is returned.

If y is > 0 and x is 0, pi/2 is returned.

If x is 0, a pole error does not occur.

If either *x* or *y* is NaN, a NaN is returned.

If the result underflows, a range error may occur and y/x is returned.

If y is 0 and x is -0, $\pm x$ is returned.

If y is 0 and x is +0, 0 is returned.

For finite values of $\pm y > 0$, if x is $-\ln f$, $\pm x$ is returned.

For finite values of $\pm y > 0$, if x is +Inf, 0 is returned.

For finite values of x, if y is $\pm \ln f$, $\pm x/2$ is returned.

If y is $\pm \ln x$ and x is $-\ln x$, $\pm 3\pi x$ is returned.

If y is $\pm \ln x$ and x is $+\ln x$, $\pm \sin 4$ is returned.

If both arguments are 0, a domain error does not occur.

Related Information

math.h in AIX 5L Version 5.2 Files Reference.

atan, atanf, or atanl Subroutine

Purpose

Computes the arc tangent.

Syntax

```
#include <math.h>
float atanf (x)
float x;
long double at an (x)
long double x;
double atan (x)
double x;
```

Description

The **atanf**, **atanl**, and **atan** subroutines compute the principal value of the arc tangent of the x parameter.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these functions. On return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the atanf, atanl, and atan subroutines return the arc tangent of x in the range [-pi /2, pi/2] radians.

If x is NaN, a NaN is returned.

If x is 0, x is returned.

If x is $\pm \ln f$, $\pm x/2$ is returned.

If x is subnormal, a range error may occur and x is returned.

Related Information

The "atan2f, atan2l, or atan2 Subroutine" on page 73 and "atanh, atanhf, or atanhl Subroutine."

math.h in AIX 5L Version 5.2 Files Reference.

atanh, atanhf, or atanhl Subroutine

Purpose

Computes the inverse hyperbolic tangent.

Syntax

```
#include <math.h>
float atanhf (x)
float x;
long double atanhl (x)
long double x;
double atanh (x)
double x;
```

Description

The **atanhf**, **atanhl**, and **atanh** subroutines compute the inverse hyperbolic tangent of the *x* parameter.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE ALL EXCEPT) before calling these functions. On return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the atanhf, atanhl, and atanh subroutines return the inverse hyperbolic tangent of the given argument.

If x is ± 1 , a pole error occurs, and **atanhf**, **atanhf**, and **atanh** return the value of the macro HUGE_VALF, HUGE VALL, and HUGE VAL respectively, with the same sign as the correct value of the function.

For finite |x|>1, a domain error occurs, and a NaN is returned.

If x is NaN, a NaN is returned.

If x is 0, x is returned.

If x is $\pm \ln f$, a domain error shall occur, and a NaN is returned.

If x is subnormal, a range error may occur and x is returned.

Error Codes

The atanhf, atanhI, and atanh subroutines return NaNQ and set errno to EDOM if the absolute value of x is greater than 1.

Related Information

"exp, expf, or expl Subroutine" on page 205

math.h in AIX 5L Version 5.2 Files Reference.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

atof atoff Subroutine

Purpose

Converts an ASCII string to a floating-point or double floating-point number.

Libraries

Standard C Library (libc.a)

Syntax

#include <stdlib.h> double atof (NumberPointer) const char *NumberPointer; float atoff (NumberPointer) char *NumberPointer;

Description

The **atof** subroutine converts a character string, pointed to by the *NumberPointer* parameter, to a double-precision floating-point number. The atoff subroutine converts a character string, pointed to by the NumberPointer parameter, to a single-precision floating-point number. The first unrecognized character ends the conversion.

Except for behavior on error, the atof subroutine is equivalent to the strtod subroutine call, with the EndPointer parameter set to (char**) NULL.

Except for behavior on error, the atoff subroutine is equivalent to the strtof subroutine call, with the EndPointer parameter set to (char**) NULL.

These subroutines recognize a character string when the characters are in one of two formats: numbers or numeric symbols.

- For a string to be recognized as a number, it should contain the following pieces in the following order:
 - 1. An optional string of white-space characters
 - 2. An optional sign
 - 3. A nonempty string of digits optionally containing a radix character
 - 4. An optional exponent in E-format or e-format followed by an optionally signed integer.
- · For a string to be recognized as a numeric symbol, it should contain the following pieces in the following order:
 - 1. An optional string of white-space characters
 - 2. An optional sign
 - 3. One of the strings: INF, infinity, NaNQ, NaNS, or NaN (case insensitive)

The atoff subroutine is not part of the ANSI C Library. These subroutines are at least as accurate as required by the IEEE Standard for Binary Floating-Point Arithmetic. The atof subroutine accepts at least 17 significant decimal digits. The atoff and subroutine accepts at least 9 leading 0's. Leading 0's are not counted as significant digits.

Parameters

NumberPointer Specifies a character string to convert.

EndPointer Specifies a pointer to the character that ended the scan or a null value.

Return Values

Upon successful completion, the **atof**, and **atoff** subroutines return the converted value. If no conversion could be performed, a value of 0 is returned and the **errno** global variable is set to indicate the error.

Error Codes

If the conversion cannot be performed, a value of 0 is returned, and the **errno** global variable is set to indicate the error.

If the conversion causes an overflow (that is, the value is outside the range of representable values), +/HUGE_VAL is returned with the sign indicating the direction of the overflow, and the errno global variable is set to ERANGE.

If the conversion would cause an underflow, a properly signed value of 0 is returned and the **errno** global variable is set to **ERANGE**.

The **atoff** subroutine has only one rounding error. (If the **atof** subroutine is used to create a double-precision floating-point number and then that double-precision number is converted to a floating-point number, two rounding errors could occur.)

Related Information

The scanf subroutine, atol, or atoi subroutine, wstrtol, watol, or watoi subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

128-Bit long double Floating-Point Format in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

atol or atoll Subroutine

Purpose

Converts a string to a long integer.

Syntax

```
#include <stdlib.h>
long long atol1 (nptr)
const char *nptr;
long atol (nptr)
const char *nptr;
```

Description

The **atoll** and **atol** subroutines (str) are equivalent to strtoll(nptr, (char **)NULL, 10) and strtol(nptr, (char **)NULL, 10), respectively. If the value cannot be represented, the behavior is undefined.

Parameters

nptr Points to the string to be converted into a long integer.

Return Values

The atoll and atol subroutines return the converted value if the value can be represented.

Related Information

strtol, strtoll, strtoll, strtoll, or atoi Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

audit Subroutine

Purpose

Enables and disables system auditing.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/audit.h>
int audit ( Command, Argument)
int Command;
int Argument;
```

Description

The audit subroutine enables or disables system auditing.

When auditing is enabled, audit records are created for security-relevant events. These records can be collected through the auditbin ("auditbin Subroutine" on page 81) subroutine, or through the /dev/audit special file interface.

Parameters

Command

Defined in the sys/audit.h file, can be one of the following values:

AUDIT_QUERY

Returns a mask indicating the state of the auditing subsystem. The mask is a logical ORing of the AUDIT_ON, AUDIT_OFF, and AUDIT_PANIC flags. The Argument parameter is ignored.

AUDIT_ON

Enables auditing. If auditing is already enabled, only the failure-mode behavior changes. The Argument parameter specifies recovery behavior in the event of failure and may be either 0 or the value AUDIT_PANIC.

Note: If AUDIT_PANIC is specified, bin-mode auditing must be enabled before the audit subroutine call.

AUDIT_OFF

Disables the auditing system if auditing is enabled. If the auditing system is disabled, the **audit** subroutine does nothing. The *Argument* parameter is ignored.

AUDIT RESET

Disables the auditing system (as does **AUDIT OFF**) and resets the auditing system. If auditing is already disabled, only the system configuration is reset. Resetting the audit configuration involves clearing the audit events and audited objects table, and terminating bin and stream auditing. The Argument parameter is ignored.

AUDIT EVENT THRESHOLD

Audit event records will be buffered until a total of Argument records have been saved, at which time the audit event records will be flushed to disk. An Argument value of zero disables this functionality. This parameter only applies to AIX 4.1.4 and later.

AUDIT BYTE THRESHOLD

Audit event data will be buffered until a total of Argument bytes of data have been saved, at which time the audit event data will be flushed to disk. An Argument value of zero disables this functionality. This parameter only applies to AIX 4.1.4 and later.

Specifies the behavior when a bin write fails (for AUDIT ON) or specifies the size of the audit event buffer (for AUDIT EVENT THRESHOLD and AUDIT BYTE THRESHOLD). For all other commands, the value of **Argument** is ignored. The valid values are:

AUDIT PANIC

The operating system shuts down if an audit record cannot be written to a bin. Note: If AUDIT_PANIC is specified, bin-mode auditing must be enabled before the audit subroutine call.

BufferSize

The number of bytes or audit event records which will be buffered. This parameter is valid only with the command AUDIT_BYTE_THRESHOLD and

AUDIT EVENT THRESHOLD. A value of zero will disable either byte (for AUDIT_BYTE_THRESHOLD) or event (for AUDIT_EVENT_THRESHOLD) buffering.

Return Values

For a Command value of AUDIT QUERY, the audit subroutine returns, upon successful completion, a mask indicating the state of the auditing subsystem. The mask is a logical ORing of the AUDIT ON, AUDIT_OFF, AUDIT_PANIC, and AUDIT_NO_PANIC flags. For any other Command value, the audit subroutine returns 0 on successful completion.

If the audit subroutine fails, a value of -1 is returned and the errno global variable is set to indicate the error.

Argument

Error Codes

The **audit** subroutine fails if either of the following is true:

EINVAL The Command parameter is not one of AUDIT_ON, AUDIT_OFF, AUDIT_RESET, or

AUDIT_QUERY.

EINVAL The Command parameter is AUDIT_ON and the Argument parameter specifies values other than

AUDIT_PANIC.

EPERM The calling process does not have root user authority.

Files

dev/audit Specifies the audit pseudo-device from which the audit records are read.

Related Information

The **auditbin** ("auditbin Subroutine") subroutine, **auditevents** ("auditevents Subroutine" on page 83) subroutine, **auditlog** ("auditlog Subroutine" on page 85) subroutine, **auditobj** ("auditobj Subroutine" on page 86) subroutine, **auditproc** ("auditproc Subroutine" on page 90) subroutine.

The audit command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

auditbin Subroutine

Purpose

Defines files to contain audit records.

Library

Standard C Library (libc.a)

Syntax

#include <sys/audit.h>

```
int auditbin (Command, Current, Next, Threshold)
int Command;
int Current;
int Next;
int Threshold;
```

Description

The **auditbin** subroutine establishes an audit bin file into which the kernel writes audit records. Optionally, this subroutine can be used to establish an overflow bin into which records are written when the current bin reaches the size specified by the *Threshold* parameter.

Parameters

Command

If nonzero, this parameter is a logical ORing of the following values, which are defined in the sys/audit.h file:

AUDIT EXCL

Requests exclusive rights to the audit bin files. If the file specified by the Current parameter is not the kernel's current bin file, the auditbin subroutine fails immediately with the errno variable set to EBUSY.

AUDIT_WAIT

The auditbin subroutine should not return until:

bin full The kernel writes the number of bytes specified by the Threshold parameter to the file descriptor specified by the Current parameter. Upon successful completion, the auditbin subroutine returns a 0. The kernel writes subsequent audit records to the file descriptor specified by the Next parameter.

bin failure

An attempt to write an audit record to the file specified by the Current parameter fails. If this occurs, the auditbin subroutine fails with the errno variable set to the return code from the auditwrite subroutine.

bin contention

Another process has already issued a successful call to the auditbin subroutine. If this occurs, the auditbin subroutine fails with the errno variable set to EBUSY.

system shutdown

The auditing system was shut down. If this occurs, the auditbin subroutine fails with the errno variable set to EINTR.

Current Next

A file descriptor for a file to which the kernel should immediately write audit records.

Specifies the file descriptor that will be used as the current audit bin if the value of the Threshold parameter is exceeded or if a write to the current bin fails. If this value is -1, no switch occurs.

Threshold

Specifies the maximum size of the current bin. If 0, the auditing subsystem will not switch bins. If it is nonzero, the kernel begins writing records to the file specified by the Next parameter, if writing a record to the file specified by the Cur parameter would cause the size of this file to exceed the number of bytes specified by the Threshold parameter. If no next bin is defined and AUDIT_PANIC was specified when the auditing subsystem was enabled, the system is shut down. If the size of the Threshold parameter is too small to contain a bin header and a bin tail, the auditbin subroutine fails and the errno variable is set to EINVAL.

Return Values

If the auditbin subroutine is successful, a value of 0 returns.

If the auditbin subroutine fails, a value of -1 returns and the errno global variable is set to indicate the error. If this occurs, the result of the call does not indicate whether any records were written to the bin.

Error Codes

The **auditbin** subroutine fails if any of the following is true:

EBADF The Current parameter is not a file descriptor for a regular file open for writing, or the Next

parameter is neither -1 nor a file descriptor for a regular file open for writing.

EBUSY The Command parameter specifies AUDIT_EXCL and the kernel is not writing audit records to

the file specified by the Current parameter.

EBUSY The Command parameter specifies AUDIT_WAIT and another process has already registered a

EINTR The auditing subsystem is shut down. EINVAL The Command parameter specifies a nonzero value other than AUDIT_EXCL or AUDIT_WAIT.

EINVAL The *Threshold* parameter value is less than the size of a bin header and trailer.

EPERM The caller does not have root user authority.

Related Information

The **audit** ("audit Subroutine" on page 79) subroutine, **auditevents** ("auditevents Subroutine") subroutine, **auditlog** ("auditlog Subroutine" on page 85) subroutine, **auditobj** ("auditobj Subroutine" on page 86) subroutine, **auditproc** ("auditproc Subroutine" on page 90) subroutine.

The audit command.

The audit file format.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

auditevents Subroutine

Purpose

Gets or sets the status of system event auditing.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/audit.h>
int auditevents ( Command, Classes, NClasses)
int Command;
struct audit_class *Classes;
int NClasses;
```

Description

The **auditevents** subroutine queries or sets the audit class definitions that control event auditing. Each audit class is a set of one or more audit events.

System auditing need not be enabled before calling the **auditevents** subroutine. The **audit** ("audit Subroutine" on page 79)subroutine can be directed with the **AUDIT_RESET** command to clear all event lists.

Parameters

Command

Specifies whether the event lists are to be queried or set. The values, defined in the **sys/audit.h** file, for the *Command* parameter are:

AUDIT SET

Sets the lists of audited events after first clearing all previous definitions.

AUDIT_GET

Queries the lists of audited events.

AUDIT_LOCK

Queries the lists of audited events. This value also blocks any other process attempting to set or lock the list of audit events. The lock is released when the process holding the lock dies or calls the **auditevents** subroutine with the *Command* parameter set to **AUDIT SET**.

Classes

Specifies the array of a_event structures for the AUDIT_SET operation, or after an AUDIT_GET or AUDIT_LOCK operation. The audit_class structure is defined in the sys/audit.h file and contains the following members:

ae name

A pointer to the name of the audit class.

ae list

A pointer to a list of null-terminated audit event names for this audit class. The list is ended by a null name (a leading null byte or two consecutive null bytes).

Note: Event and class names are limited to 15 significant characters.

ae_len The length of the event list in the ae_list member. This length includes the terminating null bytes. On an AUDIT_SET operation, the caller must set this member to indicate the actual length of the list (in bytes) pointed to by ae_list. On an AUDIT_GET or AUDIT_LOCK operation, the auditevents subroutine sets this member to indicate the actual size of the list.

NClasses

Serves a dual purpose. For **AUDIT_SET**, the *NClasses* parameter specifies the number of elements in the events array. For **AUDIT_GET** and **AUDIT_LOCK**, the *NClasses* parameter specifies the size of the buffer pointed to by the *Classes* parameter.

Attention: Only 32 audit classes are supported. One class is implicitly defined by the system to include all audit events (ALL). The administrator of your system should not attempt to define more than 31 audit classes.

Security

The calling process must have root user authority in order to use the auditevents subroutine.

Return Codes

If the **auditevents** subroutine completes successfully, the number of audit classes is returned if the *Command* parameter is **AUDIT_GET** or **AUDIT_LOCK**. A value of 0 is returned if the *Command* parameter is **AUDIT_SET**. If this call fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The auditevents subroutine fails if one or more of the following are true:

EPERM The calling process does not have root user authority.

EINVAL The value of Command is not AUDIT_SET, AUDIT_GET, or AUDIT_LOCK.

EINVAL The Command parameter is AUDIT_SET, and the value of the NClasses parameter is

greater than or equal to 32.

EINVAL A class name or event name is longer than 15 significant characters.

ENOSPC The value of *Command* is **AUDIT_GET** or **AUDIT_LOCK** and the size of the buffer

specified by the NClasses parameter is not large enough to hold the list of event structures

and names. If this occurs, the first word of the buffer is set to the required buffer size.

EFAULT The *Classes* parameter points outside of the process' address space.

EFAULT The ae list member of one or more **audit_class** structures passed for an **AUDIT_SET**

operation points outside of the process' address space.

EFAULT The Command value is **AUDIT_GET** or **AUDIT_LOCK** and the size of the Classes buffer is

not large enough to hold an integer.

EBUSY Another process has already called the auditevents subroutine with AUDIT_LOCK.

ENOMEM Memory allocation failed.

Related Information

The **audit** ("audit Subroutine" on page 79) subroutine, **auditbin** ("auditbin Subroutine" on page 81) subroutine, **auditlog** ("auditlog Subroutine") subroutine, **auditobj** ("auditobj Subroutine" on page 86) subroutine, **auditproc** ("auditproc Subroutine" on page 90) subroutine, **auditread** ("auditread, auditread_r Subroutines" on page 92) subroutine, **auditwrite** ("auditwrite Subroutine" on page 93) subroutine.

The audit command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

auditlog Subroutine

Purpose

Appends an audit record to the audit trail file.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/audit.h>
int auditlog ( Event, Result, Buffer, BufferSize)
char *Event;
int Result;
char *Buffer;
int BufferSize;
```

Description

The **auditlog** subroutine generates an audit record. The kernel audit-logging component appends a record for the specified *Event* if system auditing is enabled, process auditing is not suspended, and the *Event* parameter is in one or more of the audit classes for the current process.

The audit logger generates the audit record by adding the *Event* and *Result* parameters to the audit header and including the resulting information in the *Buffer* parameter as the audit tail.

Parameters

Event

The name of the audit event to be generated. This parameter should be the name of an audit event. Audit event names are truncated to 15 characters plus null.

Result Describes the result of this event. Valid values are defined in the sys/audit.h file and include

the following:

AUDIT_OK

The event was successful.

AUDIT_FAIL

The event failed.

AUDIT FAIL ACCESS

The event failed because of any access control denial.

AUDIT FAIL DAC

The event failed because of a discretionary access control denial.

AUDIT FAIL PRIV

The event failed because of a privilege control denial.

AUDIT FAIL AUTH

The event failed because of an authentication denial.

Other nonzero values of the Result parameter are converted into the AUDIT_FAIL value.

Points to a buffer containing the tail of the audit record. The format of the information in this

buffer depends on the event name.

BufferSize Specifies the size of the Buffer parameter, including the terminating null.

Return Values

Buffer

Upon successful completion, the auditlog subroutine returns a value of 0. If auditlog fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

The auditlog subroutine does not return any indication of failure to write the record where this is due to inappropriate tailoring of auditing subsystem configuration files or user-written code. Accidental omissions and typographical errors in the configuration are potential causes of such a failure.

Error Codes

The **auditlog** subroutine fails if any of the following are true:

The *Event* or *Buffer* parameter points outside of the process' address space. **EFAULT**

EINVAL The auditing system is either interrupted or not initialized. EINVAL The length of the audit record is greater than 32 kilobytes.

EPERM The process does not have root user authority.

ENOMEM Memory allocation failed.

Related Information

The audit ("audit Subroutine" on page 79) subroutine, auditbin ("auditbin Subroutine" on page 81) subroutine, auditevents ("auditevents Subroutine" on page 83) subroutine, auditobj ("auditobj Subroutine") subroutine, auditproc ("auditproc Subroutine" on page 90) subroutine, auditwrite ("auditwrite Subroutine" on page 93) subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

auditobj Subroutine

Purpose

Gets or sets the auditing mode of a system data object.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/audit.h>
int auditobj ( Command, Obj_Events, ObjSize)
int Command;
struct o_event *Obj_Events;
int ObjSize;
```

Description

The auditobj subroutine queries or sets the audit events to be generated by accessing selected objects. For each object in the file system name space, it is possible to specify the event generated for each access mode. Using the auditobi subroutine, an administrator can define new audit events in the system that correspond to accesses to specified objects. These events are treated the same as system-defined events.

System auditing need not be enabled to set or query the object audit events. The audit subroutine can be directed with the AUDIT RESET command to clear the definitions of object audit events.

Parameters

Command

Specifies whether the object audit event lists are to be read or written. The valid values, defined in the sys/audit.h file, for the Command parameter are:

AUDIT_SET

Sets the list of object audit events, after first clearing all previous definitions.

AUDIT_GET

Queries the list of object audit events.

AUDIT_LOCK

Queries the list of object audit events and also blocks any other process attempting to set or lock the list of audit events. The lock is released when the process holding the lock dies or calls the auditobj subroutine with the Command parameter set to AUDIT_SET.

Obj_Events

Specifies the array of **o_event** structures for the **AUDIT_SET** operation or for after the **AUDIT_GET** or **AUDIT_LOCK** operation. The **o_event** structure is defined in the **sys/audit.h** file and contains the following members:

o_type Specifies the type of the object, in terms of naming space. Currently, only one object-naming space is supported:

AUDIT_FILE

Denotes the file system naming space.

o_name Specifies the name of the object.

o event

Specifies any array of event names to be generated when the object is accessed. Note that event names are currently limited to 16 bytes, including the trailing null. The index of an event name in this array corresponds to an access mode. Valid indexes are defined in the **audit.h** file and include the following:

- AUDIT_READ
- AUDIT_WRITE
- AUDIT_EXEC

Note: The C++ compiler will generate a warning indicating that **o_event** is defined both as a structure and a field within that structure. Although the **o_event** field can be used within C++, the warning can by bypassed by defining **O_EVENT_RENAME**. This will replace the **o_event** field with **o_event_array**. **o_event** is the default field.

ObiSize

For an **AUDIT_SET** operation, the *ObjSize* parameter specifies the number of object audit event definitions in the array pointed to by the *Obj_Events* parameter. For an **AUDIT_GET** or **AUDIT_LOCK** operation, the *ObjSize* parameter specifies the size of the buffer pointed to by the *Obj_Events* parameter.

Return Values

If the **auditobj** subroutine completes successfully, the number of object audit event definitions is returned if the *Command* parameter is **AUDIT_GET** or **AUDIT_LOCK**. A value of 0 is returned if the *Command* parameter is **AUDIT_SET**. If this call fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditobj** subroutine fails if any of the following are true:

EFAULT The *Obj_Events* parameter points outside the address space of the process.

EFAULT The Command parameter is **AUDIT_SET**, and one or more of the o_name members points

outside the address space of the process.

EFAULT The Command parameter is **AUDIT_GET** or **AUDIT_LOCK**, and the buffer size of the

Obj_Events parameter is not large enough to hold the integer.

The value of the *Command* parameter is not **AUDIT_SET**, **AUDIT_GET** or **AUDIT_LOCK**.

The *Command* parameter is **AUDIT_SET**, and the value of one or more of the otype

members is not AUDIT_FILE.

EINVAL An event name was longer than 15 significant characters.

ENOENT The Command parameter is **AUDIT_SET**, and the parent directory of one of the file-system

objects does not exist.

ENOSPC The value of the Command parameter is AUDIT_GET or AUDIT_LOCK, and the size of the

buffer as specified by the *ObjSize* parameter is not large enough to hold the list of event structures and names. If this occurs, the first word of the buffer is set to the required buffer

size.

ENOMEM Memory allocation failed.

EBUSY Another process has called the auditobj subroutine with AUDIT_LOCK.

Related Information

The **audit** ("audit Subroutine" on page 79) subroutine, **auditbin** ("auditbin Subroutine" on page 81) subroutine, **auditevents** ("auditevents Subroutine" on page 83) subroutine, **auditlog** ("auditlog Subroutine" on page 85) subroutine, **auditproc** ("auditproc Subroutine" on page 90) subroutine.

The audit command.

The audit.h file.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

auditpack Subroutine

Purpose

Compresses and uncompresses audit bins.

Library

Security Library (libc.a)

Syntax

```
#include <sys/audit.h>
#include <stdio.h>

char *auditpack ( Expand, Buffer)
int Expand;
char *Buffer;
```

Description

The auditpack subroutine can be used to compress or uncompress bins of audit records.

Parameters

Expand

Specifies the operation. Valid values, as defined in the **sys/audit.h** header file, are one of the following:

AUDIT PACK

Performs standard compression on the audit bin.

AUDIT_UNPACK

Unpacks the compressed audit bin.

Buffer

Specifies the buffer containing the bin to be compressed or uncompressed. This buffer must contain a standard bin as described in the **audit.h** file.

Return Values

If the **auditpack** subroutine is successful, a pointer to a buffer containing the processed audit bin is returned. If unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The auditpack subroutine fails if one or more of the following values is true:

The Expand parameter is not one of the valid values (AUDIT_PACK or AUDIT_UNPACK).

The Expand parameter is AUDIT_UNPACK and the packed data in Buffer does not unpack to its

original size.

EINVAL The Expand parameter is AUDIT_PACK and the bin in the Buffer parameter is already

compressed, or the Expand parameter is AUDIT_UNPACK and the bin in the Buffer parameter

is already unpacked.

ENOSPC The **auditpack** subroutine is unable to allocate space for a new buffer.

Related Information

The auditread ("auditread, auditread_r Subroutines" on page 92) subroutine.

The auditcat command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

auditproc Subroutine

Purpose

Gets or sets the audit state of a process.

Library

Standard C Library (libc.a)

Syntax

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

```
int auditproc (ProcessID, Command, Argument, Length)
int ProcessID;
int Command;
char * Argument;
int Length;
```

Description

The **auditproc** subroutine queries or sets the auditing state of a process. There are two parts to the auditing state of a process:

- The list of classes to be audited for this process. Classes are defined by the auditevents ("auditevents Subroutine" on page 83) subroutine. Each class includes a set of audit events. When a process causes an audit event, that event may be logged in the audit trail if it is included in one or more of the audit classes of the process.
- The audit status of the process. Auditing for a process may be suspended or resumed. Functions that generate an audit record can first check to see whether auditing is suspended. If process auditing is suspended, no audit events are logged for a process. For more information, see the **auditlog** ("auditlog Subroutine" on page 85) subroutine.

Parameters

ProcessID The process ID of the process to be affected. If ProcessID is 0, the auditproc subroutine

affects the current process.

Command The action to be taken. Defined in the audit.h file, valid values include:

AUDIT KLIST EVENTS

Sets the list of audit classes to be audited for the process and also sets the user's default audit classes definition within the kernel. The Argument parameter is a pointer to a list of null-terminated audit class names. The Length parameter is the length of this list, including null bytes.

AUDIT QEVENTS

Returns the list of audit classes defined for the current process if ProcessID is 0. Otherwise, it returns the list of audit classes defined for the specified process ID. The Argument parameter is a pointer to a character buffer. The Length parameter specifies the size of this buffer. On return, this buffer contains a list of null-terminated audit class names. A null name terminates the list.

AUDIT EVENTS

Sets the list of audit classes to be audited for the process. The *Argument* parameter is a pointer to a list of null-terminated audit class names. The Length parameter is the length of this list, including null bytes.

AUDIT QSTATUS

Returns the audit status of the current process. You can only check the status of the current process. If the *ProcessID* parameter is nonzero, a -1 is returned and the errno global variable is set to EINVAL. The Length and Argument parameters are ignored. A return value of AUDIT_SUSPEND indicates that auditing is suspended. A return value of AUDIT_RESUME indicates normal auditing for this process.

AUDIT STATUS

Sets the audit status of the current process. The Length parameter is ignored, and the *ProcessID* parameter must be zero. If *Argument* is **AUDIT_SUSPEND**, the audit status is set to suspend event auditing for this process. If the Argument parameter is **AUDIT_RESUME**, the audit status is set to resume event auditing for this process.

Argument A character pointer for the audit class buffer for an AUDIT EVENT or AUDIT QEVENTS value

of the Command parameter or an integer defining the audit status to be set for an

AUDIT_STATUS operation.

Size of the audit class character buffer. Length

Return Values

The **auditproc** subroutine returns the following values upon successful completion:

- The previous audit status (AUDIT_SUSPEND or AUDIT_RESUME), if the call queried or set the audit status (the Command parameter specified AUDIT QSTATUS) or AUDIT STATUS)
- · A value of 0 if the call queried or set audit events (the Command parameter specified **AUDIT QEVENTS** or **AUDIT EVENTS**)

Error Codes

If the **auditproc** subroutine fails if one or more of the following are true:

EINVAL An invalid value was specified for the *Command* parameter.

EINVAL The Command parameter is set to the AUDIT_QSTATUS or AUDIT_STATUS value and the

pid value is nonzero.

EINVAI The Command parameter is set to the AUDIT_STATUS value and the Argument parameter

is not set to AUDIT_SUSPEND or AUDIT_RESUME.

ENOSPC The Command parameter is AUDIT_QEVENTS, and the buffer size is insufficient. In this

case, the first word of the Argument parameter is set to the required size.

EFAULT The Command parameter is AUDIT_QEVENTS or AUDIT_EVENTS and the Argument

parameter points to a location outside of the process' allocated address space.

ENOMEM Memory allocation failed.

EPERM The caller does not have root user authority.

Related Information

The audit ("audit Subroutine" on page 79) subroutine, auditbin ("auditbin Subroutine" on page 81) subroutine, auditevents ("auditevents Subroutine" on page 83) subroutine, auditlog ("auditlog Subroutine" on page 85) subroutine, auditobi ("auditobi Subroutine" on page 86) subroutine, auditwrite ("auditwrite Subroutine" on page 93) subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

auditread, auditread r Subroutines

Purpose

Reads an audit record.

Library

Security Library (libc.a)

Syntax

```
#include <sys/audit.h>
#include <stdio.h>
char *auditread ( FilePointer, AuditRecord)
FILE *FilePointer;
struct aud rec *AuditRecord;
char *auditread r ( FilePointer, AuditRecord, RecordSize, StreamInfo)
FILE *FilePointer;
struct aud rec *AuditRecord;
size t RecordSize;
void **StreamInfo;
```

Description

The auditread subroutine reads the next audit record from the specified file descriptor. Bins on this input stream are unpacked and uncompressed if necessary.

The auditread subroutine can not be used on more than one FilePointer as the results can be unpredictable. Use the **auditread r** subroutine instead.

The auditread r subroutine reads the next audit from the specified file descriptor. This subroutine is thread safe and can be used to handle multiple open audit files simultaneously by multiple threads of execution.

The auditread_r subroutine is able to read multiple versions of audit records. The version information contained in an audit record is used to determine the correct size and format of the record. When an input record header is larger than AuditRecord, an error is returned. In order to provide for binary compatibility with previous versions, if RecordSize is the same size as the original (struct aud rec), the input record is converted to the original format and returned to the caller.

Parameters

FilePointer Specifies the file descriptor from which to read.

AuditRecord Specifies the buffer to contain the header. The first short in this buffer must contain a valid

number for the header.

RecordSize The size of the buffer referenced by AuditRecord.

StreamInfo A pointer to an opaque datatype used to hold information related to the current value of

FilePointer. For each new value of FilePointer, a new StreamInfo pointer must be used. StreamInfo must be initialized to NULL by the user and is initialized by **auditread_r** when first used. When FilePointer has been closed, the value of StreamInfo can be passed to

the free subroutine to be deallocated.

Return Values

If the **auditread** subroutine completes successfully, a pointer to a buffer containing the tail of the audit record is returned. The length of this buffer is returned in the ah_length field of the header file. If this subroutine is unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The auditread subroutine fails if one or more of the following is true:

EBADF The *FilePointer* value is not valid.

ENOSPC The **auditread** subroutine is unable to allocate space for the tail buffer.

Other error codes are returned by the read subroutine.

Related Information

The auditpack ("auditpack Subroutine" on page 89) subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

auditwrite Subroutine

Purpose

Writes an audit record.

Library

Security Library (libc.a)

Syntax

```
#include <sys/audit.h>
#include <stdio.h>

int auditwrite (Event, Result, Buffer1, Length1, Buffer2, Length2, ...)
char * Event;
int Result;
char * Buffer1, *Buffer2 ...;
int Length1, Length2 ...;
```

Description

The **auditwrite** subroutine builds the tail of an audit record and then writes it with the **auditlog** subroutine. The tail is built by gathering the specified buffers. The last buffer pointer must be a null.

If the **auditwrite** subroutine is to be called from a program invoked from the **inittab** file, the **setpcred** subroutine should be called first to establish the process' credentials.

Parameters

Event Specifies the name of the event to be logged.

Result Specifies the audit status of the event. Valid values are defined in the sys/audit.h file

and are listed in the auditlog subroutine.

Buffer1, Buffer2 Specifies the character buffers containing audit tail information. Note that numerical

values must be passed by reference. The correct size can be computed with the

sizeof C function.

Length1, Length2 Specifies the lengths of the corresponding buffers.

Return Values

If the **auditwrite** subroutine completes successfully, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The auditwrite subroutine fails if the following is true:

ENOSPC The **auditwrite** subroutine is unable to allocate space for the tail buffer.

Other error codes are returned by the auditlog subroutine.

Related Information

The auditlog ("auditlog Subroutine" on page 85) subroutine, setpcred subroutine.

The inittab file.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

authenticate Subroutine

Purpose

Verifies a user's name and password.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
```

```
int authenticate (UserName, Response, Reenter, Message)
char *UserName;
char *Response;
int *Reenter;
char **Message;
```

Description

The **authenticate** subroutine maintains requirements users must satisfy to be authenticated to the system. It is a recallable interface that prompts for the user's name and password. The user must supply a character string at the prompt issued by the *Message* parameter. The *Response* parameter returns the user's response to the **authenticate** subroutine. The calling program makes no assumptions about the number of prompt messages the user must satisfy for authentication.

The *Reenter* parameter remains a nonzero value until the user satisfies all prompt messages or answers incorrectly. Once the *Reenter* parameter is zero, the return code signals whether authentication passed or failed.

The **authenticate** subroutine ascertains the authentication domains the user can attempt. The subroutine reads the **SYSTEM** line from the user's stanza in the **/etc/security/user** file. Each token that appears in the **SYSTEM** line corresponds to a method that can be dynamically loaded and processed. Likewise, the system can provide multiple or alternate authentication paths.

The **authenticate** routine maintains internal state information concerning the next prompt message presented to the user. If the calling program supplies a different user name before all prompts are complete for the user, the internal state information is reset and prompt messages begin again.

If the user has no defined password, or the **SYSTEM** grammar explicitly specifies no authentication required, the user is not required to respond to any prompt messages. Otherwise, the user is always initially prompted to supply a password.

The **authenticate** subroutine can be called initially with the cleartext password in the *Response* parameter. If the user supplies a password during the initial invocation but does not have a password, authentication fails. If the user wants the **authenticate** subroutine to supply a prompt message, the *Response* parameter is a null pointer on initial invocation.

The **authenticate** subroutine sets the **AUTHSTATE** environment variable used by name resolution subroutines, such as the **getpwnam** subroutine. This environment variable indicates the registry to which to user authenticated. Values for the **AUTHSTATE** environment variable include **DCE**, **compat**, and token names that appear in a **SYSTEM** grammar. A null value can exist if the **cron** daemon or other utilities that do not require authentication is called.

Parameters

UserName Points to the user's name that is to be authenticated.

Response Specifies a character string containing the user's response to an authentication prompt.

Reenter Points to a Boolean value that signals whether the **authenticate** subroutine has completed

processing. If the *Reenter* parameter is a nonzero value, the **authenticate** subroutine expects the user to satisfy the prompt message provided by the *Message* parameter. If the *Reenter* parameter is

0, the authenticate subroutine has completed processing.

Message Points to a pointer that the **authenticate** subroutine allocates memory for and fills in. This string is

suitable for printing and issues prompt messages (if the *Reenter* parameter is a nonzero value). It also issues informational messages such as why the user failed authentication (if the *Reenter*

parameter is 0). The calling application is responsible for freeing this memory.

Return Values

Upon successful completion, the **authenticate** subroutine returns a value of 0. If this subroutine fails, it returns a value of 1.

Error Codes

The authenticate subroutine is unsuccessful if one of the following values is true:

ENOENT Indicates that the user is unknown to the system.

ESAD Indicates that authentication is denied. EINVAL Indicates that the parameters are not valid. ENOMEM Indicates that memory allocation (malloc) failed.

Note: The DCE mechanism requires credentials on successful authentication that apply only to the authenticate process and its children.

Related Information

The **ckuserID** ("ckuserID Subroutine" on page 137) subroutine.

basename Subroutine

Purpose

Return the last element of a path name.

Library

Standard C Library (libc.a)

Syntax

#include <libgen.h>

char *basename (char *path)

Description

Given a pointer to a character string that contains a path name, the basename subroutine deletes trailing "/" characters from path, and then returns a pointer to the last component of path. The "/" character is defined as trailing if it is not the first character in the string.

If path is a null pointer or points to an empty string, a pointer to a static constant "." is returned.

Return Values

The **basename** function returns a pointer to the last component of *path*.

The basename function returns a pointer to a static constant "." if path is a null pointer or points to an empty string.

The **basename** function may modify the string pointed to by *path* and may return a pointer to static storage that may then be overwritten by a subsequent call to the basename subroutine.

Examples

Input string	Output string	
"/usr/lib"	"lib"	
"/usr/"	"usr"	
"/"	"/"	

Related Information

The dirname ("dirname Subroutine" on page 173) subroutine.

bcopy, bcmp, bzero or ffs Subroutine

Purpose

Performs bit and byte string operations.

Library

Standard C Library (libc.a)

Syntax

#include <strings.h>

void bcopy (Source, Destination, Length) const void *Source, char *Destination; size t Length;

int bcmp (String1, String2, Length) const void *String1, *String2; size_t Length;

void bzero (String, Length) char *String; int Length;

int ffs (Index) int Index;

Description

Note: The bcopy subroutine takes parameters backwards from the strcpy subroutine.

The bcopy, bcmp, and bzero subroutines operate on variable length strings of bytes. They do not check for null bytes as do the string routines.

The **bcopy** subroutine copies the value of the *Length* parameter in bytes from the string in the *Source* parameter to the string in the Destination parameter.

The **bcmp** subroutine compares the byte string in the *String1* parameter against the byte string of the String2 parameter, returning a zero value if the two strings are identical and a nonzero value otherwise. Both strings are assumed to be *Length* bytes long.

The **bzero** subroutine zeroes out the string in the *String* parameter for the value of the *Length* parameter in bytes.

The **ffs** subroutine finds the first bit set in the *Index* parameter passed to it and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates that the value passed is 0.

Related Information

The memcmp, memcpy, memchr, memcpy, memmove, memset ("memccpy, memchr, memcmp, memcpy, memset or memmove Subroutine" on page 683) subroutines, strcat, strncat, strxfrm, strcpy, strncpy, or strdup subroutine, strcmp, strncasecmp, strncasecmp, or strcoll subroutine, strlen, strrchr, strrchr, strpbrk, strspn, strcspn, strstr, or strtok subroutine, swab subroutine.

List of String Manipulation Subroutines and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

bessel: j0, j1, jn, y0, y1, or yn Subroutine

Purpose

Computes Bessel functions.

Libraries

```
IEEE Math Library (libm.a)
or System V Math Library (libmsaa.a)
```

Syntax

```
#include <math.h>
double j0 (x)
double x;
double j1 (x)
double x;

double jn (n, x)
int n;
double x;
double y0 (x)
double x;
double y1 (x)
double x;
double yn (n, x)
int n;
double x;
```

Description

Bessel functions are used to compute wave variables, primarily in the field of communications.

The **j0** subroutine and **j1** subroutine return Bessel functions of x of the first kind, of orders 0 and 1, respectively. The **jn** subroutine returns the Bessel function of x of the first kind of order n.

The y0 subroutine and y1 subroutine return the Bessel functions of x of the second kind, of orders 0 and 1, respectively. The yn subroutine returns the Bessel function of x of the second kind of order n. The value of x must be positive.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **-Im** flag. To compile the **j0.c** file, for example:

```
cc j0.c -lm
```

Parameters

- Specifies some double-precision floating-point value.
- Specifies some integer value.

Return Values

When using libm.a (-lm), if x is negative, y0, y1, and yn return the value NaNQ. If x is 0, y0, y1, and ynreturn the value -HUGE VAL.

When using libmsaa.a (-Imsaa), values too large in magnitude cause the functions j0, j1, y0, and y1 to return 0 and to set the errno global variable to ERANGE. In addition, a message indicating TLOSS error is printed on the standard error output.

Nonpositive values cause y0, y1, and yn to return the value -HUGE and to set the errno global variable to **EDOM**. In addition, a message indicating argument DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the matherr subroutine when using libmsaa.a (-Imsaa).

Related Information

The **matherr** ("matherr Subroutine" on page 665) subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

bindprocessor Subroutine

Purpose

Binds kernel threads to a processor.

Library

Standard C library (libc.a)

Syntax

```
#include <sys/processor.h>
int bindprocessor (What, Who, Where)
int What;
int Who;
cpu t Where;
```

Description

The bindprocessor subroutine binds a single kernel thread, or all kernel threads in a process, to a processor, forcing the bound threads to be scheduled to run on that processor. It is important to understand that a process itself is not bound, but rather its kernel threads are bound. Once kernel threads are bound, they are always scheduled to run on the chosen processor, unless they are later unbound. When a new thread is created, it has the same bind properties as its creator. This applies to the initial thread in the new process created by the fork subroutine: the new thread inherits the bind properties of the thread which called fork. When the exec subroutine is called, thread properties are left unchanged.

The **bindprocessor** subroutine will fail if the target process has a *Resource Attachment*.

Programs that use processor bindings should become Dynamic Logical Partitioning (DLPAR) aware. Refer to Dynamic Logical Partitioning in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs* for more information.

Parameters

What Specifies whether a process or a thread is being bound to a processor. The What parameter can

take one of the following values:

BINDPROCESS

A process is being bound to a processor.

BINDTHREAD

A thread is being bound to a processor.

Who Indicates a process or thread identifier, as appropriate for the What parameter, specifying the

process or thread which is to be bound to a processor.

Where If the Where parameter is a bind CPU identifier, it specifies the processor to which the process or

thread is to be bound. A value of PROCESSOR_CLASS_ANY unbinds the specified process or

thread, which will then be able to run on any processor.

The sysconf subroutine can be used to retrieve information about the number of online processors

in the system.

Return Values

On successful completion, the **bindprocessor** subroutine returns 0. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **bindprocessor** subroutine is unsuccessful if one of the following is true:

EINVAL The *What* parameter is invalid, or the *Where* parameter indicates an invalid processor number

or a processor class which is not currently available.

ESRCH The specified process or thread does not exist.

EPERM The caller does not have root user authority, and the *Who* parameter specifies either a process,

or a thread belonging to a process, having a real or effective user ID different from that of the

calling process. The target process has a Resource Attachment.

Related Information

The **bindprocessor** command.

The **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutine, **fork** ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, **sysconf** subroutine, **thread_self** subroutine.

Controlling Processor Use and Dynamic Logical Partitioning in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

brk or sbrk Subroutine

Purpose

Changes data segment space allocation.

Library

Standard C Library (libc.a)

Syntax

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

int brk (EndDataSegment) char *EndDataSegment;

void *sbrk (Increment) intptr_t Increment;

Description

The brk and sbrk subroutines dynamically change the amount of space allocated for the data segment of the calling process. (For information about segments, see the exec subroutine. For information about the maximum amount of space that can be allocated, see the ulimit and getrlimit subroutines.)

The change is made by resetting the break value of the process, which determines the maximum space that can be allocated. The break value is the address of the first location beyond the current end of the data region. The amount of available space increases as the break value increases. The available space is initialized to a value of 0 at the time it is used. The break value can be automatically rounded up to a size appropriate for the memory management architecture.

The **brk** subroutine sets the break value to the value of the *EndDataSegment* parameter and changes the amount of available space accordingly.

The **sbrk** subroutine adds to the break value the number of bytes contained in the *Increment* parameter and changes the amount of available space accordingly. The Increment parameter can be a negative number, in which case the amount of available space is decreased.

Parameters

Specifies the effective address of the maximum available data. **EndDataSegment**

Increment Specifies any integer.

Return Values

Upon successful completion, the **brk** subroutine returns a value of 0, and the **sbrk** subroutine returns the old break value. If either subroutine is unsuccessful, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The brk subroutine and the sbrk subroutine are unsuccessful and the allocated space remains unchanged if one or more of the following are true:

ENOMEM The requested change allocates more space than is allowed by a system-imposed

maximum. (For information on the system-imposed maximum on memory space, see the

ulimit system call.)

ENOMEM The requested change sets the break value to a value greater than or equal to the start

address of any attached shared-memory segment. (For information on shared memory

operations, see the **shmat** subroutine.)

Related Information

The **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutines, getrlimit ("getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit Subroutine" on page 356) subroutine, **shmat** subroutine, **shmdt** subroutine, **ulimit** subroutine.

The _end ("_end, _etext, or _edata Identifier" on page 184), _etext ("_end, _etext, or _edata Identifier" on page 184), or edata (" end, etext, or edata Identifier" on page 184) identifier.

Subroutine Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

bsearch Subroutine

Purpose

Performs a binary search.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
void *bsearch (Key, Base, NumberOfElements, Size, ComparisonPointer)
const void *Key;
const void *Base;
size t NumberOfElements;
size t Size;
int (*ComparisonPointer) (const void *, const void *);
```

Description

The **bsearch** subroutine is a binary search routine.

The **bsearch** subroutine searches an array of *NumberOfElements* objects, the initial member of which is pointed to by the *Base* parameter, for a member that matches the object pointed to by the *Key* parameter. The size of each member in the array is specified by the *Size* parameter.

The array must already be sorted in increasing order according to the provided comparison function ComparisonPointer parameter.

Parameters

Key Points to the object to be sought in the array. Points to the element at the base of the table. Base NumberOfElements Specifies the number of elements in the array.

Points to the comparison function, which is called with two arguments that point to ComparisonPointer

the Key parameter object and to an array member, in that order.

Size Specifies the size of each member in the array.

Return Values

If the Key parameter value is found in the table, the **bsearch** subroutine returns a pointer to the element found.

If the Key parameter value is not found in the table, the **bsearch** subroutine returns the null value. If two members compare as equal, the matching member is unspecified.

For the ComparisonPointer parameter, the comparison function compares its parameters and returns a value as follows:

- If the first parameter is less than the second parameter, the ComparisonPointer parameter returns a value less than 0.
- If the first parameter is equal to the second parameter, the ComparisonPointer parameter returns a value of 0.
- If the first parameter is greater than the second parameter, the ComparisonPointer parameter returns a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

The Key and Base parameters should be of type pointer-to-element and cast to type pointer-to-character. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Related Information

The **hsearch** ("hsearch, hcreate, or hdestroy Subroutine" on page 425) subroutine, **Isearch** ("Isearch or Ifind Subroutine" on page 644) subroutine, **gsort** subroutine.

Knuth, Donald E.; The Art of Computer Programming, Volume 3. Reading, Massachusetts, Addison-Wesley, 1981.

Searching and Sorting Example Program and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

btowc Subroutine

Purpose

Single-byte to wide-character conversion.

Library

Standard Library (libc.a)

Syntax

```
#include <stdio.h>
#include <wchar.h>
wint t btowc (intc);
```

Description

The btowc function determines whether c constitutes a valid (one-byte) character in the initial shift state.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values

The btowc function returns WEOF if c has the value EOF or if (unsigned char) c does not constitute a valid (one-byte) character in the initial shift state. Otherwise, it returns the wide-character representation of that character.

Related Information

The wctob subroutine.

check lock Subroutine

Purpose

Conditionally updates a single word variable atomically.

Library

Standard C library (libc.a)

Syntax

```
#include <sys/atomic_op.h>
boolean_t _check_lock ( word addr, old val, new val)
atomic_p word addr;
int old val;
int new val;
```

Parameters

word_addr Specifies the address of the single word variable.

old_val Specifies the old value to be checked against the value of the single word variable. new_val Specifies the new value to be conditionally assigned to the single word variable.

Description

The **check lock** subroutine performs an atomic (uninterruptible) sequence of operations. The compare_and_swap subroutine is similar, but does not issue synchronization instructions and therefore is inappropriate for updating lock words.

Note: The word variable must be aligned on a full word boundary.

Return Values

FALSE Indicates that the single word variable was equal to the old value and has been set to the new

TRUE Indicates that the single word variable was not equal to the old value and has been left

unchanged.

Related Information

The _clear_lock ("_clear_lock Subroutine") subroutine.

_clear_lock Subroutine

Purpose

Stores a value in a single word variable atomically.

Library

Standard C library (libc.a)

Syntax

#include <sys/atomic op.h>

```
void _clear_lock ( word addr, value)
atomic_p word addr;
int value
```

Parameters

word_addr Specifies the address of the single word variable. value Specifies the value to store in the single word variable.

Description

The **clear lock** subroutine performs an atomic (uninterruptible) sequence of operations.

This subroutine has no return values.

Note: The word variable must be aligned on a full word boundary.

Related Information

The _check_lock ("_check_lock Subroutine" on page 104) subroutine.

cabs, cabsf, or cabsl Subroutine

Purpose

Returns a complex absolute value.

Syntax

```
#include <complex.h>
double cabs (z)
double complex z;
float cabsf (z)
float complex z;
long double cabsl (z)
long double complex z;
```

Description

The cabs, cabsf, or cabsl subroutines compute the complex absolute value (also called norm, modulus, or magnitude) of the z parameter.

Parameters

Specifies the value to be computed.

Return Values

Returns the complex absolute value.

cacos, cacosf, or cacosl Subroutine

Purpose

Computes the complex arc cosine.

Syntax

```
#include <complex.h>
double complex cacos (z)
double complex z;
float complex cacosf (z)
float complex z;
long double complex cacosl (z)
long double complex z;
```

Description

The cacos, cacosf, or cacosl subroutine computes the complex arc cosine of z, with branch cuts outside the interval [-1, +1] along the real axis.

Parameters

Z

Specifies the value to be computed.

Return Values

The cacos, cacosf, or cacosl subroutine returns the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0, pi] along the real axis.

cacosh, cacoshf, or cacoshl Subroutines

Purpose

Computes the complex arc hyperbolic cosine.

Syntax

```
#include <complex.h>
double complex cacosh (z)
double complex z;
float complex cacoshf (z)
float complex z;
long double complex cacoshl (z)
long double complex z;
```

Description

The cacosh, cacosh, or cacosh subroutine computes the complex arc hyperbolic cosine of the z parameter, with a branch cut at values less than 1 along the real axis.

Parameters

Z

Specifies the value to be computed.

Return Values

The cacosh, cacoshf, or cacoshl subroutine returns the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis and in the interval [-i pi , +i pi] along the imaginary axis.

Related Information

The "ccosh, ccoshf, or ccoshl Subroutine" on page 115.

carg, cargf, or cargl Subroutine

Purpose

Returns the complex argument value.

Syntax

```
#include <complex.h>
double carg (z)
double complex z;
float cargf (z)
float complex z;
long double cargl (z)
long double complex z;
```

Description

The carg, cargf, or cargl subroutine computes the argument (also called phase angle) of the z parameter, with a branch cut along the negative real axis.

Parameters

Specifies the value to be computed.

Return Values

The carg, cargf, or cargl subroutine returns the value of the argument in the interval [-pi, +pi].

Related Information

The "cimag, cimagf, or cimagl Subroutine" on page 134, "conj, conjf, or conjl Subroutine" on page 148, and "cproj, cprojf, or cprojl Subroutine" on page 156.

casin, casinf, or casinl Subroutine

Purpose

Computes the complex arc sine.

Syntax

```
#include <complex.h>
double complex casin (z)
double complex z;
float complex casinf (z)
float complex z;
long double complex casinl (z)
long double complex z;
```

Description

The casin, casinf, or casinI subroutine computes the complex arc sine of the z parameter, with branch cuts outside the interval [-1, +1] along the real axis.

Parameters

Specifies the value to be computed.

Return Values

The casin, casinf, or casinl subroutine returns the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-pi/2, +pi/2] along the real axis.

Related Information

The "csin, csinf, or csinl Subroutine" on page 159.

casinh, casinfh, or casinlh Subroutine

Purpose

Computes the complex arc hyperbolic sine.

Syntax

```
#include <complex.h>
double complex casinh (z)
double complex z;
float complex casinhf (z)
float complex z;
long double complex casinhl (z)
long double complex z;
```

Description

The **casinh**, **casinfh**, and **casinlh** subroutines compute the complex arc hyperbolic sine of the z parameter, with branch cuts outside the interval [-i, +i] along the imaginary axis.

Parameters

Specifies the value to be computed.

Return Values

The casinh, casinfh, and casinlh subroutines return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval [-i pi/2, +i pi/2] along the imaginary axis.

Related Information

The "casin, casinf, or casinl Subroutine" on page 107.

catan, catanf, or catanl Subroutine

Purpose

Computes the complex arc tangent.

Syntax

```
#include <complex.h>
double complex catan (z)
double complex z;
float complex catanf (z)
float complex z;
long double complex catanl (z)
long double complex z;
```

Description

The catan, catanf, and catanl subroutines compute the complex arc tangent of z, with branch cuts outside the interval [-i, +i] along the imaginary axis.

Parameters

Specifies the value to be computed.

Return Values

The catan, catanf, and catanl subroutines return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-pi/2, +pi/2] along the real axis.

Related Information

"catanh, catanhf, or catanhl Subroutine"

catanh, catanhf, or catanhl Subroutine

Purpose

Computes the complex arc hyperbolic tangent.

Syntax

```
#include <complex.h>
double complex catanh (z)
double complex z;
float complex catanhf (z)
float complex z;
long double complex catanhl (z)
long double complex z;
```

Description

The **catanh**, **catanh**, and **catanh** subroutines compute the complex arc hyperbolic tangent of z, with branch cuts outside the interval [-1, +1] along the real axis.

Parameters

Specifies the value to be computed. 7

Return Values

The catanh, catanhf, and catanhl subroutines return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval [-i pi/2, +i pi/2] along the imaginary axis.

Related Information

"catan, catanf, or catanl Subroutine" on page 109

catclose Subroutine

Purpose

Closes a specified message catalog.

Library

Standard C Library (libc.a)

Syntax

#include <nl types.h>

int catclose (CatalogDescriptor) nl_catd CatalogDescriptor;

Description

The catclose subroutine closes a specified message catalog. If your program accesses several message catalogs and you reach the maximum number of opened catalogs (specified by the NL MAXOPEN constant), you must close some catalogs before opening additional ones. If you use a file descriptor to implement the **nl** catd data type, the catclose subroutine closes that file descriptor.

The catclose subroutine closes a message catalog only when the number of calls it receives matches the total number of calls to the **catopen** subroutine in an application. All message buffer pointers obtained by prior calls to the catgets subroutine are not valid when the message catalog is closed.

Parameters

CatalogDescriptor

Points to the message catalog returned from a call to the **catopen** subroutine.

Return Values

The catclose subroutine returns a value of 0 if it closes the catalog successfully, or if the number of calls it receives is fewer than the number of calls to the **catopen** subroutine.

The **catclose** subroutine returns a value of -1 if it does not succeed in closing the catalog. The **catclose** subroutine is unsuccessful if the number of calls it receives is greater than the number of calls to the catopen subroutine, or if the value of the CatalogDescriptor parameter is not valid.

Related Information

The catgets ("catgets Subroutine") subroutine, catopen ("catopen Subroutine" on page 112) subroutine.

For more information about the Message Facility, see Message Facility Overview for Programming in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

For more information about subroutines and libraries, see Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

catgets Subroutine

Purpose

Retrieves a message from a catalog.

Library

Standard C Library (libc.a)

Syntax

#include <nl types>

```
char *catgets (CatalogDescriptor, SetNumber, MessageNumber, String)
nl catd CatalogDescriptor;
int SetNumber, MessageNumber;
const char * String;
```

Description

The catgets subroutine retrieves a message from a catalog after a successful call to the catopen subroutine. If the catgets subroutine finds the specified message, it loads it into an internal character string buffer, ends the message string with a null character, and returns a pointer to the buffer.

The catgets subroutine uses the returned pointer to reference the buffer and display the message. However, the buffer can not be referenced after the catalog is closed.

Parameters

CatalogDescriptor Specifies a catalog description that is returned by the catopen subroutine.

SetNumber Specifies the set ID.

MessageNumber Specifies the message ID. The SetNumber and MessageNumber parameters

specify a particular message to retrieve in the catalog.

String Specifies the default character-string buffer.

Return Values

If the catgets subroutine is unsuccessful for any reason, it returns the user-supplied default message string specified by the String parameter.

Related Information

The catclose ("catclose Subroutine" on page 110) subroutine, catopen ("catopen Subroutine" on page 112) subroutine.

For more information about the Message Facility, see Message Facility Overview for Programming in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

For more information about subroutines and libraries, see Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

catopen Subroutine

Purpose

Opens a specified message catalog.

Library

Standard C Library (libc.a)

Syntax

```
#include <nl_types.h>
nl catd catopen (CatalogName, Parameter)
const char *CatalogName;
int Parameter;
```

Description

The **catopen** subroutine opens a specified message catalog and returns a catalog descriptor used to retrieve messages from the catalog. The contents of the catalog descriptor are complete when the catgets subroutine accesses the message catalog. The nl_catd data type is used for catalog descriptors and is defined in the nl_types.h file.

If the catalog file name referred to by the CatalogName parameter contains a leading / (slash), it is assumed to be an absolute path name. If the catalog file name is not an absolute path name, the user environment determines which directory paths to search. The NLSPATH environment variable defines the directory search path. When this variable is used, the setlocale subroutine must be called before the catopen subroutine.

A message catalog descriptor remains valid in a process until that process or a successful call to one of the exec functions closes it.

You can use two special variables, %N and %L, in the NLSPATH environment variable. The %N variable is replaced by the catalog name referred to by the call that opens the message catalog. The %L variable is replaced by the value of the LC_MESSAGES category.

The value of the LC MESSAGES category can be set by specifying values for the LANG, LC ALL, or LC MESSAGES environment variable. The value of the LC MESSAGES category indicates which locale-specific directory to search for message catalogs. For example, if the catopen subroutine specifies a catalog with the name mycmd, and the environment variables are set as follows:

```
NLSPATH=../%N:./%N:/system/nls/%L/%N:/system/nls/%N LANG=fr FR
```

then the application searches for the catalog in the following order:

```
../mycmd
./mycmd
/system/nls/fr FR/mycmd
/system/nls/mycmd
```

If you omit the %N variable in a directory specification within the NLSPATH environment variable, the application assumes that it defines a catalog name and opens it as such and will not traverse the rest of the search path.

If the NLSPATH environment variable is not defined, the catopen subroutine uses the default path. See the /etc/environment file for the NLSPATH default path. If the LC MESSAGES category is set to the default value C, and the LC__FASTMSG environment variable is set to true, then subsequent calls to the catgets subroutine generate pointers to the program-supplied default text.

The catopen subroutine treats the first file it finds as a message file. If you specify a non-message file in a NLSPATH, for example, /usr/bin/ls, catopen treats /usr/bin/ls as a message catalog. Thus no messages are found and default messages are returned. If you specify /tmp in a NLSPATH, /tmp is opened and searched for messages and default messages are displayed.

Parameters

CatalogName Parameter

Specifies the catalog file to open.

Determines the environment variable to use in locating the message catalog. If the value of the *Parameter* parameter is 0, use the **LANG** environment variable without regard to the LC MESSAGES category to locate the catalog. If the value of the Parameter parameter is the NL_CAT_LOCALE macro, use the LC_MESSAGES category to locate

the catalog.

Return Values

The catopen subroutine returns a catalog descriptor. If the LC MESSAGES category is set to the default value C, and the LC FASTMSG environment variable is set to true, the catopen subroutine returns a value of -1.

If the LC MESSAGES category is not set to the default value C but the categor subroutine returns a value of -1, an error has occurred during creation of the structure of the nl_catd data type or the catalog name referred to by the CatalogName parameter does not exist.

Related Information

The catclose ("catclose Subroutine" on page 110) subroutine, catgets ("catgets Subroutine" on page 111) subroutine, exec ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutines. setlocale subroutine.

The **environment** file.

For more information about the Message Facility, see the Message Facility Overview for Programming in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

For more information about subroutines and libraries, see the Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

cbrtf, cbrtl, or cbrt Subroutine

Purpose

Computes the cube root.

Syntax

```
#include <math.h>
float cbrtf (x)
float x;
long double cbrtl (x)
long double x;
double cbrt (x)
double x;
```

Description

The **cbrtf**, **cbrtl**, and **cbrt** subroutines compute the real cube root of the *x* argument.

Parameters

x Specifies the value to be computed.

Return Values

Upon successful completion, the cbrtf, cbrtl, and cbrt subroutines return the cube root of x.

If x is NaN, an NaN is returned.

If x is ± 0 or $\pm \ln x$, x is returned.

Related Information

math.h in AIX 5L Version 5.2 Files Reference.

ccos, ccosf, or ccosl Subroutine

Purpose

Computes the complex cosine.

Syntax

```
#include <complex.h>
double complex ccos (z)
double complex z;
float complex ccosf (z)
float complex z;
long double complex ccosl (z)
long double complex z;
```

Description

The **ccos**, **ccosf**, and **ccosl** subroutines compute the complex cosine of z.

Parameters

z Specifies the value to be computed.

Return Values

The **ccos**, **ccosf**, and **ccosl** subroutines return the complex cosine value.

Related Information

"cacos, cacosf, or cacosl Subroutine" on page 105

ccosh, ccoshf, or ccoshl Subroutine

Purpose

Computes the complex hyperbolic cosine.

Syntax

```
#include <complex.h>
double complex ccosh (z)
double complex z;
float complex ccoshf (z)
float complex z;
long double complex ccoshl (z)
long double complex z;
```

Description

The ccosh, ccoshf, and ccoshl subroutines compute the complex hyperbolic cosine of z.

Parameters

Specifies the value to be computed.

Return Values

The ccosh, ccoshf, and ccoshl subroutines return the complex hyperbolic cosine value.

Related Information

"cacosh, cacoshf, or cacoshl Subroutines" on page 106

ccsidtocs or cstoccsid Subroutine

Purpose

Provides conversion between coded character set IDs (CCSID) and code set names.

Library

The iconv Library (libiconv.a)

Syntax

```
#include <iconv.h>
CCSID cstoccsid (* Codeset)
const char *Codeset;
```

```
char *ccsidtocs ( CCSID)
CCSID CCSID;
```

Description

The **cstoccsid** subroutine returns the CCSID of the code set specified by the *Codeset* parameter. The **ccsidtocs** subroutine returns the code set name of the CCSID specified by *CCSID* parameter. CCSIDs are registered IBM coded character set IDs.

Parameters

Codeset Specifies the code set name to be converted to its corresponding CCSID.

Specifies the CCSID to be converted to its corresponding code set name.

Return Values

If the code set is recognized by the system, the **cstoccsid** subroutine returns the corresponding CCSID. Otherwise, null is returned.

If the CCSID is recognized by the system, the **ccsidtocs** subroutine returns the corresponding code set name. Otherwise, a null pointer is returned.

Related Information

For more information about code set conversion, see Converters Overview for Programming in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

The National Language Support Overview for Programming in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

ceil, ceilf, or ceill Subroutine

Purpose

Computes the ceiling value.

Syntax

```
#include <math.h>
float ceilf (x)
float x;
long double ceill (x)
long double x;
double ceil (x)
double x;
```

Description

The **ceilf**, **ceill**, and **ceil** subroutines compute the smallest integral value not less than x.

An application wishing to check for error situations should set the **errno** global variable to zero and call feclearexcept(FE ALL EXCEPT) before calling these functions. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the smallest integral value to be computed. X

Return Values

Upon successful completion, the ceilf, ceill, and ceil subroutines return the smallest integral value not less than x, expressed as a type float, long double, or double, respectively.

If x is NaN, a NaN is returned.

If x is ± 0 or $\pm \ln x$ is returned.

If the correct value would cause overflow, a range error occurs and the ceilf, ceill, and ceil subroutines return the value of the macro HUGE_VALF, HUGE_VALL, and HUGE_VAL, respectively.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, "floor, floorf, floorl, nearest, trunc, itrunc, or uitrunc Subroutine" on page 234, and "class, _class, finite, isnan, or unordered Subroutines" on page 138.

math.h in AIX 5L Version 5.2 Files Reference.

cexp, cexpf, or cexpl Subroutine

Purpose

Performs complex exponential computations.

Syntax

```
#include <complex.h>
double complex cexp (z)
double complex z;
float complex cexpf (z)
float complex z;
long double complex cexpl (z)
long double complex z;
```

Description

The **cexp**, **cexpf**, and **cexpl** subroutines compute the complex exponent of z, defined as e^z .

Parameters

Specifies the value to be computed.

Return Values

The cexp, cexpf, and cexpl subroutines return the complex exponential value of z.

Related Information

The "clog, clogf, or clog! Subroutine" on page 140.

cfgetospeed, cfsetospeed, cfgetispeed, or cfsetispeed Subroutine

Purpose

Gets and sets input and output baud rates.

Library

Standard C Library (libc.a)

Syntax

```
#include <termios.h>
speed t cfgetospeed ( TermiosPointer)
const struct termios *TermiosPointer;
int cfsetospeed (TermiosPointer, Speed)
struct termios *TermiosPointer;
speed t Speed;
speed t cfgetispeed (TermiosPointer)
const struct termios *TermiosPointer;
int cfsetispeed (TermiosPointer, Speed)
struct termios *TermiosPointer;
speed_t Speed;
```

Description

The baud rate subroutines are provided for getting and setting the values of the input and output baud rates in the termios structure. The effects on the terminal device described below do not become effective and not all errors are detected until the tcsetattr function is successfully called.

The input and output baud rates are stored in the **termios** structure. The supported values for the baud rates are shown in the table that follows this discussion.

The **termios.h** file defines the type **speed_t** as an unsigned integral type.

The cfgetospeed subroutine returns the output baud rate stored in the termios structure pointed to by the TermiosPointer parameter.

The **cfsetospeed** subroutine sets the output baud rate stored in the **termios** structure pointed to by the TermiosPointer parameter to the value specified by the Speed parameter.

The cfgetispeed subroutine returns the input baud rate stored in the termios structure pointed to by the TermiosPointer parameter.

The **cfsetispeed** subroutine sets the input baud rate stored in the **termios** structure pointed to by the TermiosPointer parameter to the value specified by the Speed parameter.

Certain values for speeds have special meanings when set in the termios structure and passed to the tcsetattr function. These values are discussed in the tcsetattr subroutine.

The following table lists possible baud rates:

Baud Rate Values

Name	Description	Name	Description
В0	Hang up	B600	600 baud
B5	50 baud	B1200	1200 baud
B75	75 baud	B1800	1800 baud
B110	110 baud	B2400	2400 baud
B134	134 baud	B4800	4800 baud
B150	150 baud	B9600	9600 baud
B200	200 baud	B19200	19200 baud
B300	300 baud	B38400	38400 baud

The termios.h file defines the name symbols of the table.

Parameters

TermiosPointer Points to a **termios** structure. Speed Specifies the baud rate.

Return Values

The cfgetospeed and cfgetispeed subroutines return exactly the value found in the termios data structure, without interpretation.

Both the cfsetospeed and cfsetispeed subroutines return a value of 0 if successful and -1 if unsuccessful.

Examples

To set the output baud rate to 0 (which forces modem control lines to stop being asserted), enter:

```
cfsetospeed (&my termios, B0);
tcsetattr (stdout, TCSADRAIN, &my termios);
```

Related Information

The tcsetattr subroutine.

The termios.h file.

Input and Output Handling Programmer's Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

chacl or fchacl Subroutine

Purpose

Changes the permissions on a file.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/acl.h>
#include <sys/mode.h>

int chacl ( Path, ACL, ACLSize)
char *Path;
struct acl *ACL;
int ACLSize;

int fchacl ( FileDescriptor, ACL, ACLSize)
int FileDescriptor;
struct acl *ACL;
int ACLSize;
```

Description

The **chacl** and **fchacl** subroutines set the access control attributes of a file according to the Access Control List (ACL) structure pointed to by the *ACL* parameter.

Parameters

Path

Specifies the path name of the file.

ACL

Specifies the ACL to be established on the file. The format of an ACL is defined in the sys/acl.h file and contains the following members:

acl len

Specifies the size of the ACL (Access Control List) in bytes, including the base

Note: The entire ACL for a file cannot exceed one memory page (4096 bytes).

acl mode

Specifies the file mode.

The following bits in the acl_mode member are defined in the sys/mode.h file and are significant for this subroutine:

S ISUID

Enables the setuid attribute on an executable file.

S ISGID

Enables the setgid attribute on an executable file. Enables the group-inheritance attribute on a directory.

S_ISVTX

Enables linking restrictions on a directory.

S IXACL

Enables extended ACL entry processing. If this attribute is not set, only the base entries (owner, group, and default) are used for access authorization checks.

Other bits in the mode, including the following, are ignored:

u_access

Specifies access permissions for the file owner.

g_access

Specifies access permissions for the file group.

o_access

Specifies access permissions for the default class of others.

acl ext[]

Specifies an array of the extended entries for this access control list.

The members for the base ACL (owner, group, and others) can contain the following bits, which are defined in the sys/access.h file:

R_ACC

Allows read permission.

W ACC

Allows write permission.

X_ACC Allows execute or search permission.

Specifies the file descriptor of an open file.

ACLSize Specifies the size of the buffer containing the ACL.

Note: The **chacl** subroutine requires the *Path*, *ACL*, and *ACLSize* parameters. The **fchacl** subroutine requires the FileDescriptor, ACL, and ACLSize parameters.

ACL Data Structure for chacl

FileDescriptor

Each access control list structure consists of one struct acl structure containing one or more struct acl entry structures with one or more struct ace id structures.

If the struct ace id structure has id type set to ACEID USER or ACEID GROUP, there is only one id_data element. To add multiple IDs to an ACL you must specify multiple struct ace_id structures when id_type is set to ACEID_USER or ACEID_GROUP. In this case, no error is returned for the multiple

elements, and the access checking examines only the first element. Specifically, the errno value EINVAL is not returned for acl len being incorrect in the ACL structure although more than one uid or gid is specified.

Return Values

Upon successful completion, the chacl and fchacl subroutines return a value of 0. If the chacl or fchacl subroutine fails, a value of -1 is returned, and the errno global variable is set to indicate the error.

Error Codes

The **chacl** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

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

ENOENT A component of the Path does not exist or has the disallow truncation attribute (see the

ulimit subroutine).

ENOENT The Path parameter was null.

EACCES Search permission is denied on a component of the Path prefix.

The Path parameter points to a location outside of the allocated address space of the **EFAULT**

process.

ESTALE The process' root or current directory is located in a virtual file system that has been

unmounted.

ELOOP Too many symbolic links were encountered in translating the *Path* parameter. ENOENT A symbolic link was named, but the file to which it refers does not exist.

ENAMETOOLONG A component of the Path parameter exceeded 255 characters, or the entire Path parameter

exceeded 1023 characters.

The chacl or fchacl subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

EROFS The file specified by the Path parameter resides on a read-only file system.

The ACL parameter points to a location outside of the allocated address space of the process. **EFAULT**

The ACL parameter does not point to a valid ACL. **EINVAL EINVAL** The acl len member in the ACL is not valid. **EIO** An I/O error occurred during the operation.

ENOSPC The size of the ACL parameter exceeds the system limit of one memory page (4KB).

EPERM The effective user ID does not match the ID of the owner of the file, and the invoker does not

have root user authority.

The **fchacl** subroutine fails and the file permissions remain unchanged if the following is true:

EBADF The file descriptor FileDescriptor is not valid.

If Network File System (NFS) is installed on your system, the chacl and fchacl subroutines can also fail if the following is true:

ETIMEDOUT The connection timed out.

Security

Access Control: The invoker must have search permission for all components of the Path prefix.

Auditing Events:

Event Information Path chacl

fchacl **FileDescriptor**

Related Information

The acl chg ("acl chg or acl fchg Subroutine" on page 9) subroutine, acl get ("acl get or acl fget Subroutine" on page 11) subroutine, acl_put ("acl_put or acl_fput Subroutine" on page 13) subroutine, acl set ("acl set or acl fset Subroutine" on page 15) subroutine, chmod ("chmod or fchmod Subroutine" on page 124) subroutine, stat subroutine, statacl subroutine.

The aciget command, aciput command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

chdir Subroutine

Purpose

Changes the current directory.

Library

Standard C Library (libc.a)

Syntax

#include <unistd.h>

int chdir (Path) const char *Path;

Description

The **chdir** subroutine changes the current directory to the directory indicated by the *Path* parameter.

Parameters

Path

A pointer to the path name of the directory. If the Path parameter refers to a symbolic link, the chdir subroutine sets the current directory to the directory pointed to by the symbolic link. If Network File System (NFS) is installed on the system, this path can cross into another node.

The current directory, also called the current working directory, is the starting point of searches for path names that do not begin with a / (slash). The calling process must have search access to the directory specified by the Path parameter.

Return Values

Upon successful completion, the chdir subroutine returns a value of 0. Otherwise, a value of -1 is returned and the errno global variable is set to identify the error.

Error Codes

The **chdir** subroutine fails and the current directory remains unchanged if one or more of the following are true:

EACCES Search access is denied for the named directory.

ENOENT The named directory does not exist. **ENOTDIR** The path name is not a directory.

The **chdir** subroutine can also be unsuccessful for other reasons. See Appendix A. Base Operating System Error Codes for Services That Require Path-Name Resolution (Appendix A, "Base Operating System Error Codes for Services That Require Path-Name Resolution," on page 1001) for a list of additional error codes.

If NFS is installed on the system, the chdir subroutine can also fail if the following is true:

ETIMEDOUT

The connection timed out.

Related Information

The **chroot** ("chroot Subroutine" on page 131) subroutine.

The cd command.

Appendix A, "Base Operating System Error Codes for Services That Require Path-Name Resolution," on page 1001.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

chmod or fchmod Subroutine

Purpose

Changes file access permissions.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/stat.h>
int chmod ( Path, Mode)
const char *Path;
mode t Mode;
int fchmod ( FileDescriptor, Mode)
int FileDescriptor;
mode_t Mode;
```

Description

The **chmod** subroutine sets the access permissions of the file specified by the *Path* parameter. If Network File System (NFS) is installed on your system, this path can cross into another node.

Use the **fchmod** subroutine to set the access permissions of an open file pointed to by the *FileDescriptor* parameter.

The access control information is set according to the *Mode* parameter.

Parameters

FileDescriptor Mode

Specifies the file descriptor of an open file.

Specifies the bit pattern that determines the access permissions. The Mode parameter is constructed by logically ORing one or more of the following values, which are defined in the sys/mode.h file:

S ISUID

Enables the **setuid** attribute for an executable file. A process executing this program acquires the access rights of the owner of the file.

S ISGID

Enables the **setgid** attribute for an executable file. A process executing this program acquires the access rights of the group of the file. Also, enables the group-inheritance attribute for a directory. Files created in this directory have a group equal to the group of the directory.

The following attributes apply only to files that are directly executable. They have no meaning when applied to executable text files such as shell scripts and awk scripts.

S_ISVTX

Enables the link/unlink attribute for a directory. Files cannot be linked to in this directory. Files can only be unlinked if the requesting process has write permission for the directory and is either the owner of the file or the directory.

S ISVTX

Enables the save text attribute for an executable file. The program is not unmapped after usage.

S ENFMT

Enables enforcement-mode record locking for a regular file. File locks requested with the lockf subroutine are enforced.

S IRUSR

Permits the file's owner to read it.

S IWUSR

Permits the file's owner to write to it.

S IXUSR

Permits the file's owner to execute it (or to search the directory).

S IRGRP

Permits the file's group to read it.

S IWGRP

Permits the file's group to write to it.

S IXGRP

Permits the file's group to execute it (or to search the directory).

S IROTH

Permits others to read the file.

S IWOTH

Permits others to write to the file.

S IXOTH

Permits others to execute the file (or to search the directory).

Other mode values exist that can be set with the **mknod** subroutine but not with the chmod subroutine.

Return Values

Upon successful completion, the chmod subroutine and fchmod subroutines return a value of 0. If the chmod subroutine or fchmod subroutine is unsuccessful, a value of -1 is returned, and the errno global variable is set to identify the error.

Error Codes

The chmod subroutine is unsuccessful and the file permissions remain unchanged if one of the following is true:

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

EACCES Search permission is denied on a component of the Path prefix.

EFAULT The Path parameter points to a location outside of the allocated address space of the

process.

ELOOP Too many symbolic links were encountered in translating the *Path* parameter.

ENOENT The named file does not exist.

ENAMETOOLONG A component of the Path parameter exceeded 255 characters, or the entire Path parameter

exceeded 1023 characters.

The **fchmod** subroutine is unsuccessful and the file permissions remain unchanged if the following is true:

EBADF The value of the FileDescriptor parameter is not valid.

The chmod or fchmod subroutine is unsuccessful and the access control information for a file remains unchanged if one of the following is true:

EPERM The effective user ID does not match the owner of the file, and the process does not have

appropriate privileges.

EROFS The named file resides on a read-only file system. An I/O error occurred during the operation. **EIO**

If NFS is installed on your system, the chmod and fchmod subroutines can also be unsuccessful if the following is

true:

ESTALE The root or current directory of the process is located in a virtual file system that has been

unmounted.

ETIMEDOUT The connection timed out.

Security

Access Control: The invoker must have search permission for all components of the Path prefix.

If you receive the EBUSY error, toggle the enforced locking attribute in the Mode parameter and retry your operation. The enforced locking attribute should never be used on a file that is part of the Trusted Computing Base.

Related Information

The acl_chg ("acl_chg or acl_fchg Subroutine" on page 9) subroutine, acl_get ("acl_get or acl_fget Subroutine" on page 11) subroutine, acl_put ("acl_put or acl_fput Subroutine" on page 13) subroutine, acl_set ("acl_set or acl_fset Subroutine" on page 15) subroutine, chacl ("chacl or fchacl Subroutine" on page 119) subroutine, statacl subroutine, stat subroutine.

The aciget command, aciput command, chmod command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

chown, fchown, lchown, chownx, or fchownx Subroutine

Purpose

Changes file ownership.

Library

Standard C Library (libc.a)

Syntax

```
Syntax for the chown, fchown, and lchown Subroutines:
```

```
#include <sys/types.h>
#include <unistd.h>
int chown (Path, Owner, Group)
const char *Path;
uid t Owner;
gid_t Group;
int fchown (FileDescriptor, Owner, Group)
int FileDescriptor;
uid_t Owner;
gid_t Group;
int Ichown (Path, Owner, Group)
const char *fname
uid t uid
gid_tgid
Syntax for the chownx and fchownx Subroutines:
#include <sys/types.h>
#include <sys/chownx.h>
int chownx (Path, Owner, Group, Flags)
char *Path;
uid t Owner;
gid_t Group;
int Flags;
int fchownx (FileDescriptor, Owner, Group, Flags)
int FileDescriptor;
uid t Owner;
gid_t Group;
int Flags;
```

Description

The chown, chownx, fchownx, and Ichown subroutines set the file owner and group IDs of the specified file system object. Root user authority is required to change the owner of a file.

A function Ichown function sets the owner ID and group ID of the named file similarity to chown function except in the case where the named file is a symbolic link. In this case Ichown function changes the ownership of the symbolic link file itself, while chown function changes the ownership of the file or directory to which the symbolic link refers.

Parameters

FileDescriptor Specifies the file descriptor of an open file.

Specifies whether the file owner ID or group ID should be changed. This parameter is Flags

constructed by logically ORing the following values:

T_OWNER_AS_IS

Ignores the value specified by the Owner parameter and leaves the owner ID of

the file unaltered.

T_GROUP_AS_IS

Ignores the value specified by the Group parameter and leaves the group ID of

the file unaltered.

Group Specifies the new group of the file. For the **chown**, **fchown**, and **lchown** commands, if

> this value is -1, the group is not changed. (A value of -1 indicates only that the group is not changed; it does not indicate a group that is not valid. An owner or group ID cannot be invalid.) For the **chownx** and **fchownx** commands, the subroutines change the Group

to -1 if -1 is supplied for Group and T_GROUP_AS_IS is not set.

Owner Specifies the new owner of the file. For the chown, fchown, and lchown commands, if

> this value is -1, the group is not changed. (A value of -1 indicates only that the group is not changed; it does not indicate a group that is not valid. An owner or group ID cannot be invalid.) For the chownx and fchownx commands, the subroutines change the Owner

to -1 if -1 is supplied for Owner and T_OWNER_AS_IS is not set

Path Specifies the full path name of the file. If Path resolves to a symbolic link, the ownership

of the file or directory pointed to by the symbolic link is changed.

Return Values

Upon successful completion, the chown, chownx, fchown, fchownx, and lchown subroutines return a value of 0. If the chown, chownx, fchown, fchownx, or lchown subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The chown, chownx, or Ichown subroutine is unsuccessful and the owner and group of a file remain unchanged if one of the following is true:

EACCESS Search permission is denied on a component of the Path parameter.

EDQUOT The new group for the file system object cannot be set because the group's quota of disk

blocks or i-nodes has been exhausted on the file system.

EFAULT The Path parameter points to a location outside of the allocated address space of the

process.

EINVAL The owner or group ID supplied is not valid.

ELOOP Too many symbolic links were encountered in translating the *Path* parameter.

ENAMETOOLONG A component of the Path parameter exceeded 255 characters, or the entire Path parameter

exceeded 1023 characters.

ENOENT A symbolic link was named, but the file to which it refers does not exist; or a component of

the Path parameter does not exist; or the process has the disallow truncation attribute set;

or the Path parameter is null.

ENOTDIR A component of the path prefix is not a directory.

EPERM The effective user ID does not match the owner of the file, and the calling process does not

have the appropriate privileges.

EROFS The named file resides on a read-only file system. **ESTALE** The root or current directory of the process is located in a virtual file system that has been

unmounted.

The **fchown** or **fchownx** subroutine is unsuccessful and the file owner and group remain unchanged if one of the following is true:

EBADF The named file resides on a read-only file system.

The new group for the file system object cannot be set because the group's guota of disk **EDQUOT**

blocks or i-nodes has been exhausted on the file system.

EIO An I/O error occurred during the operation.

Security

Access Control: The invoker must have search permission for all components of the Path parameter.

chpass Subroutine

Purpose

Changes user passwords.

Library

Standard C Library (libc.a)

Thread Safe Security Library (libs_r.a)

Syntax

```
int chpass (UserName, Response, Reenter, Message)
char *UserName;
char *Response;
int *Reenter;
char **Message;
```

Description

The **chpass** subroutine maintains the requirements that the user must meet to change a password. This subroutine is the basic building block for changing passwords and handles password changes for local, NIS, and DCE user passwords.

The *Message* parameter provides a series of messages asking for old and new passwords, or providing informational messages, such as the reason for a password change failing. The first Message prompt is a prompt for the old password. This parameter does not prompt for the old password if the user has a real user ID of 0 (zero) and is changing a local user, or if the user has no current password. The chpass subroutine does not prompt a user with root authority for an old password. It informs the program that no message was sent and that it should invoke chpass again. If the user satisfies the first Message parameter's prompt, the system prompts the user to enter the new password. Each message is contained in the Message parameter and is displayed to the user. The Response parameter returns the user's response to the chpass subroutine.

The Reenter parameter remains a nonzero value until the user satisfies all of the prompt messages or until the user incorrectly responds to a prompt message. Once the Reenter parameter is 0, the return code signals whether the password change completed or failed.

The **chpass** subroutine maintains internal state information concerning the next prompt message to present to the user. If the calling program supplies a different user name before all prompt messages are complete for the user, the internal state information is reset and prompt messages begin again.

The **chpass** subroutine determines the administration domain to use during password changes. It determines if the user is defined locally, defined in Network Information Service (NIS), or defined in Distributed Computing Environment (DCE). Password changes occur only in these domains. System administrators may override this convention with the registry value in the /etc/security/user file. If the registry value is defined, the password change can only occur in the specified domain. System administrators can use this registry value if the user is administered on a remote machine that periodically goes down. If the user is allowed to log in through some other authentication method while the server is down, password changes remain to follow only the primary server.

The **chpass** subroutine allows the user to change passwords in two ways. For normal (non-administrative) password changes, the user must supply the old password, either on the first call to the chpass subroutine or in response to the first message from chpass. If the user is root, real user ID of 0, local administrative password changes are handled by supplying a null pointer for the Response parameter during the initial call

Users that are not administered locally are always queried for their old password.

The **chpass** subroutine is always in one of the following states:

- 1. Initial state: The caller invokes the **chpass** subroutine with NULL response parameter and receives the initial password prompt in the *message* parameter.
- 2. Verify initial password: The caller invokes the chpass subroutine with the results of prompting the user with earlier message parameter as the response parameter. The caller is given a prompt to enter the new password in the *message* parameter.
- 3. Enter new password: The caller invokes the **chpass** subroutine with the results of prompting user with the new password prompt in the response parameter. The caller will be given a prompt to repeat the new password in the *message* parameter.
- 4. Verify new password: The caller invokes the **chpass** subroutine with the results of prompting the user to repeat the new password in the response parameter. The chpass subroutine then performs the actual password change.

Any step in the above process can result in the **chpass** subroutine terminating the dialog. This is signalled when the reenter variable is set to 0. The return code indicates the nature of the failure.

Parameters

UserName Specifies the user's name whose password is to be changed.

Response Specifies a character string containing the user's response to the last prompt. Points to a Boolean value used to signal whether chpass subroutine has completed Reenter

processing. If the Reenter parameter is a nonzero value, the chpass subroutine expects the

user to satisfy the prompt message provided by the Message parameter. If the Reenter

parameter is 0, the **chpass** subroutine has completed processing.

Points to a pointer that the chpass subroutine allocates memory for and fills in. This Message

> replacement string is then suitable for printing and issues challenge messages (if the Reenter parameter is a nonzero value). The string can also issue informational messages such as why the user failed to change the password (if the Reenter parameter is 0). The calling application

is responsible for freeing this memory.

Return Values

Upon successful completion, the **chpass** subroutine returns a value of 0. If the **chpass** subroutine is unsuccessful, it returns the following values:

- -1 Indicates the call failed in the thread safe library libs_r.a. ERRNO will indicate the failure code.
- 1 Indicates that the password change was unsuccessful and the user should attempt again. This return value occurs if a password restriction is not met, such as if the password is not long enough.
- Indicates that the password change was unsuccessful and the user should not attempt again. This return 2 value occurs if the user enters an incorrect old password or if the network is down (the password change cannot occur).

Error Codes

The chpass subroutine is unsuccessful if one of the following values is true:

ENOENT Indicates that the user cannot be found.

ESAD Indicates that the user did not meet the criteria to change the password. **EPERM** Indicates that the user did not have permission to change the password.

EINVAL Indicates that the parameters are not valid. **ENOMEM** Indicates that memory allocation (malloc) failed.

Related Information

The authenticate ("authenticate Subroutine" on page 94) subroutine.

chroot Subroutine

Purpose

Changes the effective root directory.

Library

Standard C Library (libc.a)

Syntax

#include <unistd.h>

int chroot (const char * Path) char *Path;

Description

The **chroot** subroutine causes the directory named by the *Path* parameter to become the effective root directory. If the Path parameter refers to a symbolic link, the chroot subroutine sets the effective root directory to the directory pointed to by the symbolic link. If Network File System (NFS) is installed on your system, this path can cross into another node.

The effective root directory is the starting point when searching for a file's path name that begins with / (slash). The current directory is not affected by the **chroot** subroutine.

The calling process must have root user authority in order to change the effective root directory. The calling process must also have search access to the new effective root directory.

The .. (double period) entry in the effective root directory is interpreted to mean the effective root directory itself. Thus, this directory cannot be used to access files outside the subtree rooted at the effective root directory.

Parameters

Path

Pointer to the new effective root directory.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **chroot** subroutine fails and the effective root directory remains unchanged if one or more of the following are true:

ENOENT The named directory does not exist. **EACCES** The named directory denies search access. **EPERM** The process does not have root user authority.

The **chroot** subroutine can be unsuccessful for other reasons. See Appendix A. Base Operating System Error Codes for Services that Require Path-Name Resolution (Appendix A, "Base Operating System Error Codes for Services That Require Path-Name Resolution," on page 1001) for a list of additional errors.

If NFS is installed on the system, the **chroot** subroutine can also fail if the following is true:

ETIMEDOUT

The connection timed out.

Related Information

The **chdir** ("chdir Subroutine" on page 123) subroutine.

The **chroot** command.

Appendix A, "Base Operating System Error Codes for Services That Require Path-Name Resolution," on page 1001.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

chssys Subroutine

Purpose

Modifies the subsystem objects associated with the SubsystemName parameter.

Library

System Resource Controller Library (libsrc.a)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
int chssys( SubsystemName, SRCSubsystem)
char *SubsystemName;
struct SRCsubsys *SRCSubsystem;
```

Description

The chssys subroutine modifies the subsystem objects associated with the specified subsystem with the values in the SRCsubsys structure. This action modifies the objects associated with subsystem in the following object classes:

- Subsystem Environment
- Subserver Type
- Notify

The Subserver Type and Notify object classes are updated only if the subsystem name has been changed.

The SRCsubsys structure is defined in the /usr/include/sys/srcobj.h file.

The program running with this subroutine must be running with the group system.

Parameters

SRCSubsystem Points to the SRCsubsys structure. SubsystemName Specifies the name of the subsystem.

Return Values

Upon successful completion, the chssys subroutine returns a value of 0. Otherwise, it returns a value of -1 and the odmerrno variable is set to indicate the error, or a System Resource Controller (SRC) error code is returned.

Error Codes

The **chssys** subroutine is unsuccessful if one or more of the following are true:

SRC_NONAME No subsystem name is specified. SRC_NOPATH No subsystem path is specified. SRC_BADNSIG Invalid stop normal signal. SRC_BADFSIG Invalid stop force signal.

SRC NOCONTACT Contact not signal, sockets, or message queues.

SRC_SSME Subsystem name does not exist. New subsystem name is already on file. SRC_SUBEXIST

SRC_SYNEXIST New subsystem synonym name is already on file. SRC_NOREC The specified **SRCsubsys** record does not exist.

SRC_SUBSYS2BIG Subsystem name is too long. SRC SYN2BIG Synonym name is too long. SRC_CMDARG2BIG Command arguments are too long. SRC_PATH2BIG Subsystem path is too long. SRC_STDIN2BIG stdin path is too long. SRC_STDOUT2BIG stdout path is too long. SRC_STDERR2BIG stderr path is too long.

Security

SRC_GRPNAM2BIG

Privilege Control: This command has the Trusted Path attribute. It has the following kernel privilege:

Group name is too long.

SET_PROC_AUDIT kernel privilege

Files Accessed:

Mode File

644 /etc/objrepos/SRCsubsys
 644 /etc/objrepos/SRCsubsvr
 644 /etc/objrepos/SRCnotify

Auditing Events:

Event Information

SRC_Chssys

Files

/etc/objrepos/SRCsubsysSRC Subsystem Configuration object class./etc/objrepos/SRCsubsvrSRC Subserver Configuration object class.

/etc/objrepos/SRCnotify SRC Notify Method object class.
/dev/SRC Specifies the AF_UNIX socket file.

/dev/.SRC-unix Specifies the location for temporary socket files.

Related Information

The **addssys** ("addssys Subroutine" on page 19) subroutine, **delssys** ("delssys Subroutine" on page 171) subroutine.

The chssys command, mkssys command, rmssys command.

System Resource Controller Overview in AIX 5L Version 5.2 System Management Concepts: Operating System and Devices.

Defining Your Subsystem to the SRC, List of SRC Subroutines, System Resource Controller (SRC) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

cimag, cimagf, or cimagl Subroutine

Purpose

Performs complex imaginary computations.

Syntax

```
#include <complex.h>
double cimag (z)
double complex z;
float cimagf (z)
float complex z;
long double cimagl (z)
long double complex z;
```

Description

The **cimag**, **cimagf**, and **cimagl** subroutines compute the imaginary part of z.

Parameters

z Specifies the value to be computed.

Return Values

The **cimag**, **cimagf**, and **cimagl** subroutines return the imaginary part value (as a real).

Related Information

"carg, cargf, or cargl Subroutine" on page 107, "conj, conjf, or conjl Subroutine" on page 148, "cproj, cprojf, or cprojl Subroutine" on page 156, and "creal, crealf, or creall Subroutine" on page 156.

ckuseracct Subroutine

Purpose

Checks the validity of a user account.

Library

Security Library (libc.a)

Syntax

```
#include <login.h>
int ckuseracct ( Name, Mode, TTY)
char *Name;
int Mode;
char *TTY;
```

Description

Note: This subroutine is obsolete and is provided only for backwards compatibility. Use the loginrestrictions subroutine, which performs a superset of the functions of the ckuseracct subroutine, instead.

The **ckuseracct** subroutine checks the validity of the user account specified by the *Name* parameter. The Mode parameter gives the mode of the account usage, and the TTY parameter defines the terminal being used for the access. The **ckuseracct** subroutine checks for the following conditions:

- · Account existence
- Account expiration

The *Mode* parameter specifies other mode-specific checks.

Parameters

Name Specifies the login name of the user whose account is to be validated. Mode

Specifies the manner of usage. Valid values as defined in the login.h file are listed below. The Mode parameter must be one of these or 0:

S_LOGIN

Verifies that local logins are permitted for this account.

Verifies that the su command is permitted and that the current process has a group ID that can invoke the **su** command to switch to the account.

S DAEMON

Verifies the account can be used to invoke daemon or batch programs using the src or cron subsystems.

S RLOGIN

Verifies the account can be used for remote logins using the **rlogind** or **telnetd** programs. Specifies the terminal of the originating activity. If this parameter is a null pointer or a null string, no TTY origin checking is done.

Security

TTY

Files Accessed:

Mode File

/etc/passwd /etc/security/user

Return Values

If the account is valid for the specified usage, the **ckuseract** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the errno global variable is set to the appropriate error code.

Error Codes

The **ckuseracct** subroutine fails if one or more of the following are true:

ENOENT The user specified in the Name parameter does not have an account.

ESTALE The user's account is expired.

EACCES The specified terminal does not have access to the specified account.

EACCES The Mode parameter is S_SU, and the current process is not permitted to use the su

command to access the specified user.

EACCES Access to the account is not permitted in the specified Mode.

EINVAL The Mode parameter is not one of S_LOGIN, S_SU, S_DAEMON, S_RLOGIN.

Related Information

The ckuserID ("ckuserID Subroutine" on page 137) subroutine, getpcred ("getpcred Subroutine" on page 341) subroutine, getpenv ("getpenv Subroutine" on page 343) subroutine, setpcred subroutine, setpenv subroutine.

The login command, rlogin command, su command, telnet command.

The **cron** daemon.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

ckuserID Subroutine

Purpose

Authenticates the user.

Note: This subroutine is obsolete and is provided for backwards compatibility. Use the authenticate ("authenticate Subroutine" on page 94) subroutine, instead.

Library

Security Library (libc.a)

Syntax

#include <login.h> int ckuserID (User, Mode) int Mode: char *User;

Description

The **ckuserID** subroutine authenticates the account specified by the *User* parameter. The mode of the authentication is given by the Mode parameter. The login and su commands continue to use the ckuserID subroutine to process the /etc/security/user auth1 and auth2 authentication methods.

The **ckuserID** subroutine depends on the **authenticate** ("authenticate Subroutine" on page 94) subroutine to process the SYSTEM attribute in the /etc/security/user file. If authentication is successful, the passwdexpired ("passwdexpired Subroutine" on page 773) subroutine is called.

Errors caused by grammar or load modules during a call to the authenticate subroutine are displayed to the user if the user was authenticated. These errors are audited with the USER_Login audit event if the user failed authentication.

Parameters

User Mode Specifies the name of the user to be authenticated.

Specifies the mode of authentication. This parameter is a bit mask and may contain one or more of the following values, which are defined in the login.h file:

S PRIMARY

The primary authentication methods defined for the *User* parameter are checked. All primary authentication checks must be passed.

S_SECONDARY

The secondary authentication methods defined for the *User* parameter are checked. Secondary authentication checks are not required to be successful.

Primary and secondary authentication methods for each user are set in the /etc/security/user file by defining the auth1 and auth2 attributes. If no primary methods are defined for a user, the SYSTEM attribute is assumed. If no secondary methods are defined, there is no default.

Security

Files Accessed:

File Mode

/etc/passwd

/etc/security/passwd r /etc/security/user /etc/security/login.cfg

Return Values

If the account is valid for the specified usage, the ckuserID subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The ckuserID subroutine fails if one or more of the following are true:

ESAD Security authentication failed for the user.

EINVAL The Mode parameter is neither S_PRIMARY nor S_SECONDARY or the Mode parameter is

both **S_PRIMARY** and **S_SECONDARY**.

Related Information

The authenticate ("authenticate Subroutine" on page 94) subroutine, ckuseracct ("ckuseracct Subroutine" on page 135) subroutine, getpcred ("getpcred Subroutine" on page 341) subroutine, getpenv ("getpenv Subroutine" on page 343) subroutine, passwdexpired ("passwdexpired Subroutine" on page 773) subroutine, setpcred subroutine, setpenv subroutine.

The **login** command, **su** command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

class, _class, finite, isnan, or unordered Subroutines

Purpose

Determines classifications of floating-point numbers.

Libraries

IEEE Math Library (libm.a) or System V Math Library (libmsaa.a)

Syntax

```
#include <math.h>
#include <float.h>
int
class(x)
double x;
#include <math.h>
#include <float.h>
int
class(x)
double x;
#include <math.h>
```

```
int finite(x)
double x;
#include <math.h>
int isnan(x)
double x;
#include <math.h>
int unordered (x, y)
double x, y;
```

Description

The class subroutine, _class subroutine, finite subroutine, isnan subroutine, and unordered subroutine determine the classification of their floating-point value. The unordered subroutine determines if a floating-point comparison involving x and y would generate the IEEE floating-point unordered condition (such as whether x or v is a NaN).

The **class** subroutine returns an integer that represents the classification of the floating-point *x* parameter. Since **class** is a reversed key word in C++. The **class** subroutine can not be invoked in a C++ program. The _class subroutine is an interface for C++ program using the class subroutine. The interface and the return value for class and class subroutines are identical. The values returned by the class subroutine are defined in the **float.h** header file. The return values are the following:

FP PLUS NORM Positive normalized, nonzero x**FP MINUS NORM** Negative normalized, nonzero x FP PLUS DENORM Positive denormalized, nonzero x FP_MINUS_DENORM Negative denormalized, nonzero x

FP_PLUS_ZERO x = +0.0FP MINUS ZERO x = -0.0FP_PLUS_INF x = +INFFP MINUS INF x = -INF

FP_NANS x =Signaling Not a Number (NaNS) FP_NANQ x = Quiet Not a Number (NaNQ)

Since class is a reserved keyword in C++, the class subroutine cannot be invoked in a C++ program. The class subroutine is an interface for the C++ program using the class subroutine. The interface and the return values for **class** and **class** subroutines are identical.

The **finite** subroutine returns a nonzero value if the x parameter is a finite number; that is, if x is not +-, INF, NaNQ, or NaNS.

The **isnan** subroutine returns a nonzero value if the x parameter is an NaNS or a NaNQ. Otherwise, it returns 0.

The **unordered** subroutine returns a nonzero value if a floating-point comparison between x and y would be unordered. Otherwise, it returns 0.

Note: Compile any routine that uses subroutines from the libm.a library with the -Im flag. To compile the class.c file, for example, enter:

```
cc class.c -lm
```

Parameters

- Specifies some double-precision floating-point value.
- Specifies some double-precision floating-point value.

Error Codes

The finite, isnan, and unordered subroutines neither return errors nor set bits in the floating-point exception status, even if a parameter is an NaNS.

Related Information

List of Numerical Manipulation Services and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

clock Subroutine

Purpose

Reports central processing unit (CPU) time used.

Library

Standard C Library (libc.a)

Syntax

#include <time.h> clock t clock (void);

Description

The **clock** subroutine reports the amount of CPU time used. The reported time is the sum of the CPU time of the calling process and its terminated child processes for which it has executed wait, system, or pclose subroutines. To measure the amount of time used by a program, the clock subroutine should be called at the beginning of the program, and that return value should be subtracted from the return value of subsequent calls to the clock subroutine. To find the time in seconds, divide the value returned by the clock subroutine by the value of the macro CLOCKS PER SEC, which is defined in the time.h file.

Return Values

The **clock** subroutine returns the amount of CPU time used.

Related Information

The getrusage, times ("getrusage, getrusage64, times, or vtimes Subroutine" on page 360) subroutine, pclose ("pclose Subroutine" on page 797) subroutine, system subroutine, vtimes ("getrusage, getrusage64, times, or vtimes Subroutine" on page 360) subroutine, wait, waitpid, wait3 subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

clog, clogf, or clogl Subroutine

Purpose

Computes the complex natural logarithm.

Syntax

```
#include <complex.h>
double complex clog (z)
double complex z;
```

```
float complex clogf (z)
float complex z;
long double complex clogl (z)
long double complex z;
```

Description

The clog, clogf, and clog subroutines compute the complex natural (base e) logarithm of z, with a branch cut along the negative real axis.

Parameters

Specifies the value to be computed.

Return Values

The clog, clogf, and clogI subroutines return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval [-i pi, +i pi] along the imaginary axis.

Related Information

"cexp, cexpf, or cexpl Subroutine" on page 117

close Subroutine

Purpose

Closes the file associated with a file descriptor.

Syntax

```
#include <unistd.h>
int close (
FileDescriptor)
int FileDescriptor;
```

Description

The close subroutine closes the file associated with the FileDescriptor parameter. If Network File System (NFS) is installed on your system, this file can reside on another node.

All file regions associated with the file specified by the FileDescriptor parameter that this process has previously locked with the lockf or fcntl subroutine are unlocked. This occurs even if the process still has the file open by another file descriptor.

If the FileDescriptor parameter resulted from an open ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine that specified O DEFER, and this was the last file descriptor, all changes made to the file since the last fsync subroutine are discarded.

If the FileDescriptor parameter is associated with a mapped file, it is unmapped. The shmat subroutine provides more information about mapped files.

The close subroutine attempts to cancel outstanding asynchronous I/O requests on this file descriptor. If the asynchronous I/O requests cannot be canceled, the application is blocked until the requests have completed.

The close subroutine is blocked until all subroutines which use the file descriptor return to usr space. For example, when a thread is calling **close** and another thread is calling **select** with the same file descriptor, the close subroutine does not return until the select call returns.

When all file descriptors associated with a pipe or FIFO special file have been closed, any data remaining in the pipe or FIFO is discarded. If the link count of the file is 0 when all file descriptors associated with the file have been closed, the space occupied by the file is freed, and the file is no longer accessible.

Note: If the FileDescriptor parameter refers to a device and the close subroutine actually results in a device close, and the device close routine returns an error, the error is returned to the application. However, the FileDescriptor parameter is considered closed and it may not be used in any subsequent calls.

All open file descriptors are closed when a process exits. In addition, file descriptors may be closed during the exec subroutine if the close-on-exec flag has been set for that file descriptor.

Parameters

FileDescriptor Specifies a valid open file descriptor.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the errno global variable is set to identify the error. If the close subroutine is interrupted by a signal that is caught, it returns a value of -1, the errno global variable is set to EINTR and the state of the FileDescriptor parameter is closed.

Error Codes

The **close** subroutine is unsuccessful if the following is true:

EBADF The FileDescriptor parameter does not specify a valid open file descriptor.

EINTR Specifies that the **close** subroutine was interrupted by a signal.

The close subroutine may also be unsuccessful if the file being closed is NFS-mounted and the server is down under the following conditions:

- The file is on a hard mount.
- · The file is locked in any manner.

The **close** subroutine may also be unsuccessful if NFS is installed and the following is true:

ETIMEDOUT The connection timed out.

Related Information

The exec ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutines, fcntl ("fcntl, dup, or dup2 Subroutine" on page 214) subroutine, ioctl ("ioctl, ioctlx, ioctl32, or ioctl32x Subroutine" on page 459) subroutine, lockfx ("lockfx, lockf, flock, or lockf64 Subroutine" on page 628) subroutine, open, openx, or creat ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine, pipe ("pipe Subroutine" on page 815) subroutine, socket subroutine.

The Input and Output Handling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

compare and swap Subroutine

Purpose

Conditionally updates or returns a single word variable atomically.

Library

Standard C library (libc.a)

Syntax

```
#include <sys/atomic op.h>
boolean t compare and swap ( word addr, old val addr, new val)
atomic p word addr;
int *old val addr;
int new val;
```

Description

The compare and swap subroutine performs an atomic operation which compares the contents of a single word variable with a stored old value. If the values are equal, a new value is stored in the single word variable and TRUE is returned; otherwise, the old value is set to the current value of the single word variable and FALSE is returned.

The compare and swap subroutine is useful when a word value must be updated only if it has not been changed since it was last read.

Note: The word containing the single word variable must be aligned on a full word boundary.

Note: If compare_and_swap is used as a locking primitive, insert an isync at the start of any critical sections.

Parameters

word_addr Specifies the address of the single word variable.

Specifies the address of the old value to be checked against (and conditionally updated old val addr

with) the value of the single word variable.

Specifies the new value to be conditionally assigned to the single word variable. new_val

Return Values

TRUE Indicates that the single word variable was equal to the old value, and has been set to the new

FALSE Indicates that the single word variable was not equal to the old value, and that its current value

has been returned in the location where the old value was previously stored.

Related Information

The fetch and add ("fetch and add Subroutine" on page 229) subroutine, fetch and and ("fetch_and_and or fetch_and_or Subroutine" on page 230) subroutine, fetch_and_or ("fetch_and_and or fetch and or Subroutine" on page 230) subroutine.

compile, step, or advance Subroutine

Purpose

Compiles and matches regular-expression patterns.

Note: Commands use the **regcomp**, **regexec**, **regfree**, and **regerror** subroutines for the functions described in this article.

Library

Standard C Library (libc.a)

Syntax

```
#define INIT declarations
#define GETC( ) getc_code
#define PEEKC() peekc code
#define UNGETC(c) ungetc code
#define RETURN(pointer) return code
#define ERROR(val) error code
#include <regexp.h>
#include <NLregexp.h>
char *compile (InString, ExpBuffer, EndBuffer, EndOfFile)
char * ExpBuffer;
char * InString, * EndBuffer;
int EndOfFile;
int step (String, ExpBuffer)
const char * String, *ExpBuffer;
int advance (String, ExpBuffer)
const char *String, *ExpBuffer;
```

Description

The /usr/include/regexp.h file contains subroutines that perform regular-expression pattern matching. Programs that perform regular-expression pattern matching use this source file. Thus, only the regexp.h file needs to be changed to maintain regular expression compatibility between programs.

The interface to this file is complex. Programs that include this file define the following six macros before the **#include <regexp.h>** statement. These macros are used by the **compile** subroutine:

INIT This macro is used for dependent declarations and initializations. It is placed right after the declaration and opening { (left brace) of the compile subroutine. The definition of the INIT buffer must end with a; (semicolon). INIT is frequently used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for the GETC, PEEKC, and UNGETC macros. Otherwise, you can use INIT to declare external variables that GETC, PEEKC, and **UNGETC** require. GETC() This macro returns the value of the next character in the regular expression pattern. Successive calls to the GETC macro should return successive characters of the pattern. PEEKC() This macro returns the next character in the regular expression. Successive calls to the PEEKC macro should return the same character, which should also be the next character returned by the GETC macro.

UNGETC(c)				

This macro causes the parameter c to be returned by the next call to the GETC and PEEKC macros. No more than one character of pushback is ever needed, and this character is guaranteed to be the last character read by the GETC macro. The return value of the UNGETC macro is always ignored.

RETURN(pointer)

This macro is used for normal exit of the compile subroutine. The pointer parameter points to the first character immediately following the compiled regular expression. This is useful for programs that have memory allocation to manage.

ERROR(val)

This macro is used for abnormal exit from the **compile** subroutine. It should never contain a **return** statement. The *val* parameter is an error number. The error values and their meanings are:

Error	Meaning
11	Interval end point too large
16	Bad number
25	\ digit out of range
36	Illegal or missing delimiter
41	No remembered search String
42	\ (?\) imbalance
43	Too many \.(
44	More than two numbers given in $\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \$
45	} expected after \.
46	First number exceeds second in $\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \$
49	[] imbalance
50	Regular expression overflow
70	Invalid endpoint in range

The **compile** subroutine compiles the regular expression for later use. The *InString* parameter is never used explicitly by the compile subroutine, but you can use it in your macros. For example, you can use the compile subroutine to pass the string containing the pattern as the InString parameter to compile and use the INIT macro to set a pointer to the beginning of this string. The example in the "Examples" on page 146 section uses this technique. If your macros do not use InString, then call compile with a value of ((char *) 0) for this parameter.

The ExpBuffer parameter points to a character array where the compiled regular expression is to be placed. The EndBuffer parameter points to the location that immediately follows the character array where the compiled regular expression is to be placed. If the compiled expression cannot fit in (*EndBuffer-ExpBuffer*) bytes, the call **ERROR**(50) is made.

The EndOfFile parameter is the character that marks the end of the regular expression. For example, in the ed command, this character is usually / (slash).

The regexp.h file defines other subroutines that perform actual regular-expression pattern matching. One of these is the **step** subroutine.

The String parameter of the step subroutine is a pointer to a null-terminated string of characters to be checked for a match.

The Expbuffer parameter points to the compiled regular expression, obtained by a call to the compile subroutine.

The **step** subroutine returns the value 1 if the given string matches the pattern, and 0 if it does not match. If it matches, then step also sets two global character pointers: loc1, which points to the first character that matches the pattern, and loc2, which points to the character immediately following the last character that matches the pattern. Thus, if the regular expression matches the entire string, loc1 points to the first character of the String parameter and loc2 points to the null character at the end of the String parameter.

The **step** subroutine uses the global variable **circf**, which is set by the **compile** subroutine if the regular expression begins with a \(^\) (circumflex). If this variable is set, **step** only tries to match the regular expression to the beginning of the string. If you compile more than one regular expression before executing the first one, save the value of circf for each compiled expression and set circf to that saved value before each call to step.

Using the same parameters that were passed to it, the step subroutine calls a subroutine named advance. The step function increments through the String parameter and calls the advance subroutine until it returns a 1, indicating a match, or until the end of String is reached. To constrain the String parameter to the beginning of the string in all cases, call the advance subroutine directly instead of calling the step subroutine.

When the advance subroutine encounters an * (asterisk) or a \{ \} sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself, trying to match the rest of the string to the rest of the regular expression. As long as there is no match, the advance subroutine backs up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. You can stop this backing-up before the initial point in the string is reached. If the **locs** global character is equal to the point in the string sometime during the backing-up process, the advance subroutine breaks out of the loop that backs up and returns 0. This is used for global substitutions on the whole line so that expressions such as s/y*//g do not loop forever.

Note: In 64-bit mode, these interfaces are not supported: they fail with a return code of 0. In order to use the 64-bit version of this functionality, applications should migrate to the **fnmatch**, **glob**, **regcomp**, and regexec functions which provide full internationalized regular expression functionality compatible with ISO 9945-1:1996 (IEEE POSIX 1003.1) and with the UNIX98 specification.

Parameters

InString Specifies the string containing the pattern to be compiled. The InString parameter is not used

explicitly by the compile subroutine, but it may be used in macros.

ExpBuffer Points to a character array where the compiled regular expression is to be placed.

EndBuffer Points to the location that immediately follows the character array where the compiled regular

expression is to be placed.

EndOfFile Specifies the character that marks the end of the regular expression. String Points to a null-terminated string of characters to be checked for a match.

Examples

The following is an example of the regular expression macros and calls:

```
#define INIT
                     register char *sp=instring;
                       (*sp++)
#define GETC()
#define PEEKC()
                        (*sp)
                       (--sp)
#define UNGETC(c)
#define RETURN(c)
                       return;
#define ERROR(c)
                      regerr()
#include <regexp.h>
compile (patstr,expbuf, &expbuf[ESIZE], '\0');
```

```
if (step (linebuf, expbuf))
  succeed( );
```

Related Information

The regcmp or regex subroutine, regcomp subroutine, regerror subroutine, regexec subroutine, regfree subroutine.

List of String Manipulation Services and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

confstr Subroutine

Purpose

Gets configurable variables.

Library

Standard C library (libc.a)

Syntax

#include <unistd.h>

size_t confstr (int name, char * buf, size_t len);

Description

The **confstr** subroutine determines the current setting of certain system parameters, limits, or options that are defined by a string value. It is mainly used by applications to find the system default value for the PATH environment variable. Its use and purpose are similar to those of the sysconf subroutine, but it returns string values rather than numeric values.

If the Len parameter is not 0 and the Name parameter has a system-defined value, the confstr subroutine copies that value into a Len-byte buffer pointed to by the Buf parameter. If the string returns a value longer than the value specified by the Len parameter, including the terminating null byte, then the confstr subroutine truncates the string to Len-1 bytes and adds a terminating null byte to the result. The application can detect that the string was truncated by comparing the value returned by the confstr subroutine with the value specified by the Len parameter.

Parameters

Specifies the system variable setting to be returned. Valid values for the Name parameter are defined Name

in the unistd.h file.

Buf Points to the buffer into which the **confstr** subroutine copies the value of the *Name* parameter.

Specifies the size of the buffer storing the value of the Name parameter. I en

Return Values

If the value specified by the Name parameter is system-defined, the confstr subroutine returns the size of the buffer needed to hold the entire value. If this return value is greater than the value specified by the Len parameter, the string returned as the Buf parameter is truncated.

If the value of the *Len* parameter is set to 0 and the *Buf* parameter is a null value, the **confstr** subroutine returns the size of the buffer needed to hold the entire system-defined value, but does not copy the string value. If the value of the *Len* parameter is set to 0 but the *Buf* parameter is not a null value, the result is unspecified.

Error Codes

The confstr subroutine will fail if:

EINVAL

The value of the name argument is invalid.

Example

To find out what size buffer is needed to store the string value of the *Name* parameter, enter: $confstr(CS_PATH, NULL, (size_t) 0)$

The **confstr** subroutine returns the size of the buffer.

Files

/usr/include/limits.h /usr/include/unistd.h Contains system-defined limits.

Contains system-defined environment variables.

Related Information

The **pathconf** ("pathconf or fpathconf Subroutine" on page 774) subroutine, **sysconf** subroutine.

The unistd.h header file.

Subroutines, Example Programs, and Libraries in *AIX 5L Version 5.2 General Programming Concepts:* Writing and Debugging Programs.

conj, conjf, or conjl Subroutine

Purpose

Computes the complex conjugate.

Syntax

```
#include <complex.h>
double complex conj (z)
double complex z;
float complex conjf (z)
float complex z;
long double complex conjl (z)
long double complex z;
```

Description

The **conj**, **conjf**, or **conjl** subroutines compute the complex conjugate of *z*, by reversing the sign of its imaginary part.

Parameters

7

Specifies the value to be computed.

Return Values

The conj, conjf, or conjl subroutines return the complex conjugate value.

Related Information

The "carg, cargf, or cargl Subroutine" on page 107, "cimag, cimagf, or cimagl Subroutine" on page 134, "cproj, cprojf, or cprojl Subroutine" on page 156, "creal, crealf, or creall Subroutine" on page 156.

conv Subroutines

Purpose

Translates characters.

Library

Standard C Library (libc.a)

Syntax

```
#include <ctype.h>
int toupper ( Character)
int Character;
int tolower (Character)
int Character;
int toupper (Character)
int Character;
int _tolower (Character)
int Character;
int toascii (Character)
int Character;
int NCesc ( Pointer, CharacterPointer)
NLchar *Pointer;
char *CharacterPointer;
int NCtoupper ( Xcharacter)
int Xcharacter;
int NCtolower (Xcharacter)
int Xcharacter;
int _NCtoupper (Xcharacter)
int Xcharacter;
int _NCtolower (Xcharacter)
int Xcharacter;
```

```
int NCtoNLchar (Xcharacter)
int Xcharacter;
int NCunesc (CharacterPointer, Pointer)
char *CharacterPointer;
NLchar *Pointer;
int NCflatchr (Xcharacter)
int Xcharacter;
```

Description

The toupper and the tolower subroutines have as domain an int, which is representable as an unsigned char or the value of EOF: -1 through 255.

If the parameter of the toupper subroutine represents a lowercase letter and there is a corresponding uppercase letter (as defined by LC CTYPE), the result is the corresponding uppercase letter. If the parameter of the tolower subroutine represents an uppercase letter, and there is a corresponding lowercase letter (as defined by LC CTYPE), the result is the corresponding lowercase letter. All other values in the domain are returned unchanged. If case-conversion information is not defined in the current locale, these subroutines determine character case according to the "C" locale.

The toupper and tolower subroutines accomplish the same thing as the toupper and tolower subroutines, but they have restricted domains. The **toupper** routine requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. The _tolower routine requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The **NC**xxxxx subroutines translate all characters, including extended characters, as code points. The other subroutines translate traditional ASCII characters only. The NCxxxxxx subroutines are obsolete and should not be used if portability and future compatibility are a concern.

The value of the Xcharacter parameter is in the domain of any legal **NLchar** data type. It can also have a special value of -1, which represents the end of file (EOF).

If the parameter of the NCtoupper subroutine represents a lowercase letter according to the current collating sequence configuration, the result is the corresponding uppercase letter. If the parameter of the NCtolower subroutine represents an uppercase letter according to the current collating sequence configuration, the result is the corresponding lowercase letter. All other values in the domain are returned unchanged.

The _NCtoupper and _NCtolower routines are macros that perform the same function as the NCtoupper and NCtolower subroutines, but have restricted domains and are faster. The _NCtoupper macro requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. The NCtolower macro requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The NCtoNLchar subroutine yields the value of its parameter with all bits turned off that are not part of an **NLchar** data type.

The NCesc subroutine converts the NLchar value of the Pointer parameter into one or more ASCII bytes stored in the character array pointed to by the CharacterPointer parameter. If the NLchar data type represents an extended character, it is converted into a printable ASCII escape sequence that uniquely identifies the extended character. NCesc returns the number of bytes it wrote. The display symbol table lists the escape sequence for each character.

The opposite conversion is performed by the **NCunesc** macro, which translates an ordinary ASCII byte or escape seguence starting at CharacterPointer into a single NLchar at Pointer. NCunesc returns the number of bytes it read.

The NCflatchr subroutine converts its parameter value into the single ASCII byte that most closely resembles the parameter character in appearance. If no ASCII equivalent exists, it converts the parameter value to a ? (question mark).

Note: The setlocale subroutine may affect the conversion of the decimal point symbol and the thousands separator.

Parameters

Character Specifies the character to be converted. Xcharacter Specifies an NLchar value to be converted. Specifies a pointer to a single-byte character array. CharacterPointer Pointer Specifies a pointer to an escape sequence.

Related Information

The Japanese conv ("Japanese conv Subroutines" on page 474) subroutines, ctype ("ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines" on page 168) subroutines, **getc, fgetc, getchar,** or **getw** ("getc, getchar, fgetc, or getw Subroutine" on page 300) subroutine, getwc, fgetwc, or getwchar ("getwc, fgetwc, or getwchar Subroutine" on page 397) subroutine, setlocale subroutine.

List of Character Manipulation Services and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

copysign, copysignf, or copysignI Subroutine

Purpose

Performs number manipulation.

Syntax

```
#include <math.h>
double copysign (x, y)
double x, double y;
float copysignf (x, y)
float x, float y;
long double copysign (x, y)
long double x, long double y;
```

Description

The copysign, copysignf, and copysignl subroutines produce a value with the magnitude of x and the sign of y.

Parameters

Specifies the magnitude. X

Specifies the sign. V

Return Values

Upon successful completion, the copysign, copysignf and copysignl subroutines return a value with a magnitude of x and a sign of y.

Related Information

signbit in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

math.h in AIX 5L Version 5.2 Files Reference.

coredump Subroutine

Purpose

Creates a **core** file without terminating the calling process.

Library

Standard C library (libc.a)

Syntax

#include <core.h>

int coredump(coredumpinfop) struct coredumpinfo *coredumpinfop ;

Description

The coredump subroutine creates a core file of the calling process without terminating the calling process. The created core file contains the memory image of the process, and this can be used with the dbx command for debugging purposes. In multithreaded processes, only one thread at a time should attempt to call this subroutine. Subsequent calls to coredump while a core dump (initiated by another thread) is in progress will fail.

Applications expected to use this facility need to be built with the -bM:UR binder flag, otherwise the routine will fail with an error code of ENOTSUP.

The coredumpinfo structure has the following fields:

Member Type	Member Name	Description
unsigned int	length	Length of the core file name
char *	name	Points to a character string that contains the name of the core file
int	reserved[8]	Reserved fields for future use

Parameters

coredumpinfop Points to the coredumpinfo structure If a NULL pointer is passed as an argument, the default file named core in the current directory is used.

Return Values

Upon successful completion, the coredump subroutine returns a value of 0. If the coredump subroutine is not successful, a value of -1 is returned and the errno global variable is set to indicate the error

Error Codes

EINVAL Invalid argument.

EACCESS Search permission is denied on a component of the path prefix, the file exists and

the pwrite permission is denied, or the file does not exist and write permission is

denied for the parent directory of the file to be created.

EINPROGRESS A core dump is already in progress.

ENOMEM Not enough memory. **ENOTSUP** Routine not supported. **EFAULT** Invalid user address.

Related Information

The adb command, dbx command.

The core file format.

cosf, cosl, or cos Subroutine

Purpose

Computes the cosine.

Syntax

```
#include <math.h>
float cosf (x)
float x;
long double cosl (x)
long double x;
double cos (x)
```

Description

double x;

The **cosf**, **cosl**, and **cos** subroutines compute the cosine of the *x*, parameter (measured in radians).

An application wishing to check for error situations should set errno to zero and call feclearexcept(FE ALL EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the $\cos f$, $\cos I$, and $\cos f$ subroutines return the cosine of f.

If x is NaN, a NaN is returned.

If x is ± 0 , the value 1.0 is returned.

If x is $\pm \ln f$, a domain error occurs, and a NaN is returned.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, _class, finite, isnan, or unordered Subroutines" on page 138.

sin, sinl, cos, cosl, tan, or tanl Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

math.h in AIX 5L Version 5.2 Files Reference.

cosh, coshf, or coshl Subroutine

Purpose

Computes the hyperbolic cosine.

Syntax

```
#include <math.h>
float coshf (x)
float x;
long double cosh1 (x)
long double x;
double cosh(x)
double x;
```

Description

The **coshf**, **coshl**, and **cosh** subroutines compute the hyperbolic cosine of the x parameter.

An application wishing to check for error situations should set errno to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these functions. On return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the **coshf**, **coshl**, and **cosh** subroutines return the hyperbolic cosine of x.

If the correct value would cause overflow, a range error occurs and the coshf, coshl, and cosh subroutines return the value of the macro HUGE VALF, HUGE VALL, and HUGE VAL, respectively. If x is NaN, a NaN is returned.

If x is ± 0 , the value 1.0 is returned.

If x is $\pm \ln f$, $+\ln f$ is returned.

Related Information

"acosh, acoshf, or acoshl Subroutine" on page 18, "feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, _class, finite, isnan, or unordered Subroutines" on page 138

sinh, sinhf, or sinhl Subroutine and tanh, tanhf, or tanhl Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

math.h in AIX 5L Version 5.2 Files Reference.

cpow, cpowf, or cpowl Subroutine

Purpose

Computes the complex power.

Syntax

```
#include <complex.h>
double complex cpow (x, y)
double complex x;
double complex y;
float complex cpowf (x, y)
float complex x;
float complex y;
long double complex cpowl (x, y)
long double complex x;
long double complex y;
```

Description

The **cpow**, **cpowf**, and **cpowl** subroutines compute the complex power function x^{\vee} , with a branch cut for the first parameter along the negative real axis.

Parameters

```
Specifies the base value.
```

Specifies the power the base value is raised to.

Return Values

The cpow, cpowf, and cpowl subroutines return the complex power function value.

Related Information

"cabs, cabsf, or cabsl Subroutine" on page 105 and "csqrt, csqrtf, or csqrtl Subroutine" on page 160

math.h in AIX 5L Version 5.2 Files Reference.

cproj, cprojf, or cprojl Subroutine

Purpose

Computes the complex projection functions.

Syntax

```
#include <complex.h>
double complex cproj (z)
double complex z;
float complex cprojf (z)
float complex z;
long double complex cprojl (z)
long double complex z;
```

Description

The **cproj**, **cprojf**, and **cprojl** subroutines compute a projection of z onto the Riemann sphere: z projects to z, except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If z has an infinite part, **cproj**(z) shall be equivalent to:

```
INFINITY + I * copysign(0.0, cimag(z))
```

Parameters

z Specifies the value to be projected.

Return Values

The cproj, cprojf, and cprojl subroutines return the value of the projection onto the Riemann sphere.

Related Information

"carg, cargf, or cargl Subroutine" on page 107, "cimag, cimagf, or cimagl Subroutine" on page 134, "conj, conjf, or conjl Subroutine" on page 148, and "creal, crealf, or creal Subroutine."

math.h in AIX 5L Version 5.2 Files Reference.

creal, crealf, or creall Subroutine

Purpose

Computes the real part of a specified value.

Syntax

```
#include <complex.h>
double creal (z)
double complex z;
float crealf (z)
float complex z;
long double creall (z)
long double complex z;
```

Description

The **creal**, **crealf**, and **creall** subroutines compute the real part of the value specified by the *z* parameter.

Parameters

Specifies the real to be computed.

Return Values

These subroutines return the real part value.

Related Information

"carg, cargf, or cargl Subroutine" on page 107, "cimag, cimagf, or cimagl Subroutine" on page 134, "conj, conjf, or conjl Subroutine" on page 148, and "cproj, cprojf, or cprojl Subroutine" on page 156

crypt, encrypt, or setkey Subroutine

Purpose

Encrypts or decrypts data.

Library

Standard C Library (libc.a)

Syntax

```
char *crypt (PW, Salt)
const char * PW, * Salt;
void encrypt (Block, EdFlag)
char Block[64];
int EdFlag;
void setkey (Key)
const char * Key;
```

Description

The crypt and encrypt subroutines encrypt or decrypt data. The crypt subroutine performs a one-way encryption of a fixed data array with the supplied PW parameter. The subroutine uses the Salt parameter to vary the encryption algorithm.

The **encrypt** subroutine encrypts or decrypts the data supplied in the *Block* parameter using the key supplied by an earlier call to the **setkey** subroutine. The data in the *Block* parameter on input must be an array of 64 characters. Each character must be an char 0 or char 1.

If you need to statically bind functions from **libc.a** for **crypt** do the following:

1. Create a file and add the following:

```
setkey
encrypt
__crypt
```

- 2. Perform the linking.
- 3. Add the following to the make file:

-bI:YourFileName

where YourFileName is the name of the file you created in step 1. It should look like the following: LDFLAGS=bnoautoimp -bI:/lib/syscalls.exp -bI:YourFileName -lc

These subroutines are provided for compatibility with UNIX system implementations.

Parameters

Block Identifies a 64-character array containing the values (char) 0 and (char) 1. Upon return, this

buffer contains the encrypted or decrypted data.

EdFlag Determines whether the subroutine encrypts or decrypts the data. If this parameter is 0, the data

> is encrypted. If this is a nonzero value, the data is decrypted. If the /usr/lib/libdes.a file does not exist and the EdFlag parameter is set to nonzero, the encrypt subroutine returns the ENOSYS

error code.

Specifies an 64-element array of 0's and 1's cast as a const char data type. The Key parameter Key

is used to encrypt or decrypt data.

PWSpecifies up to an 8-character string to be encrypted. Salt Specifies a 2-character string chosen from the following:

> A-Z Uppercase alpha characters

a-7 Lowercase alpha characters Numeric characters

Period

0-9

Slash

The Salt parameter is used to vary the hashing algorithm in one of 4096 different ways.

Return Values

The **crypt** subroutine returns a pointer to the encrypted password. The static area this pointer indicates may be overwritten by subsequent calls.

Error Codes

ENOSYS

The encrypt subroutine returns the following:

The encrypt subroutine was called with the EdFlag parameter which was set to a nonzero value. Also, the /usr/lib/libdes.a file does not exist.

Related Information

The **newpass** ("newpass Subroutine" on page 732) subroutine.

The **login** command, **passwd** command, **su** command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

csid Subroutine

Purpose

Returns the character set ID (charsetID) of a multibyte character.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
int csid ( String)
const char *String;
```

Description

The **csid** subroutine returns the charsetID of the multibyte character pointed to by the *String* parameter. No validation of the character is performed. The parameter must point to a value in the character range of the current code set defined in the current locale.

Parameters

String Specifies the character to be tested.

Return Values

Successful completion returns an integer value representing the charsetID of the character. This integer can be a number from 0 through n, where n is the maximum character set defined in the CHARSETID field of the charmap. See "Understanding the Character Set Description (charmap) Source File" in AIX 5L Version 5.2 System Management Concepts: Operating System and Devices for more information.

Related Information

The **mbstowcs** ("mbstowcs Subroutine" on page 680) subroutine, **wcsid** subroutine.

National Language Support Overview and Understanding the Character Set Description (charmap) Source File in AIX 5L Version 5.2 National Language Support Guide and Reference.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

csin, csinf, or csinl Subroutine

Purpose

Computes the complex sine.

Syntax

```
#include <complex.h>
double complex csin (z)
double complex z;
float complex csinf (z)
float complex z;
long double complex csinl (z)
long double complex z;
```

Description

The **csin**, **csinf**, and **csinI** subroutines compute the complex sine of the value specified by the *z* parameter.

Parameters

z Sn

Specifies the value to be computed.

Return Values

The csin, csinf, and csinI subroutines return the complex sine value.

Related Information

"casin, casinf, or casinl Subroutine" on page 107

csinh, csinhf, or csinhl Subroutine

Purpose

Computes the complex hyperbolic sine.

Syntax

```
#include <complex.h>
double complex csinh (z)
double complex z;
float complex csinhf (z)
float complex z;
long double complex csinhl (z)
long double complex z;
```

Description

The **csinh**, **csinhf**, and **csinhl** subroutines compute the complex hyperbolic sine of the value specified by the *z* parameter.

Parameters

7

Specifies the value to be computed.

Return Values

The **csinh**, **csinhf**, and **csinhl** subroutines return the complex hyperbolic sine value.

Related Information

"casinh, casinfh, or casinlh Subroutine" on page 108

csqrt, csqrtf, or csqrtl Subroutine

Purpose

Computes complex square roots.

Syntax

```
#include <complex.h>
double complex csqrt (z)
double complex z;
float complex csqrtf (z)
float complex z;
long double complex csqrtl (z)
long double complex z;
```

Description

The csqrt, csqrtf, and csqrtl subroutines compute the complex square root of the value specified by the z parameter, with a branch cut along the negative real axis.

Parameters

Specifies the value to be computed.

Return Values

The csqrt, csqrtf, and csqrtl subroutines return the complex square root value, in the range of the right half-plane (including the imaginary axis).

Related Information

"cabs, cabsf, or cabsl Subroutine" on page 105, "cpow, cpowf, or cpowl Subroutine" on page 155

ctan, ctanf, or ctanl Subroutine

Purpose

Computes complex tangents.

Syntax

```
#include <complex.h>
double complex ctan (z)
double complex z;
float complex ctanf (z)
float complex z;
long double complex ctanl (z)
long double complex z;
```

Description

The ctan, ctanf, and ctanl subroutines compute the complex tangent of the value specified by the z parameter.

Parameters

Specifies the value to be computed. Z

Return Values

The ctan, ctanf, and ctanl subroutines return the complex tangent value.

Related Information

"catan, catanf, or catanl Subroutine" on page 109

math.h in AIX 5L Version 5.2 Files Reference.

ctanh, ctanhf, or ctanhl Subroutine

Purpose

Computes the complex hyperbolic tangent.

Syntax

```
#include <complex.h>
double complex ctanh (z)
double complex z;
float complex ctanhf (z)
float complex z;
long double complex ctanhl (z)
long double complex z;
```

Description

The ctanh, ctanhf, and ctanhl subroutines compute the complex hyperbolic tangent of z.

Parameters

Z Specifies the value to be computed.

Return Values

The ctanh, ctanhf, and ctanhl subroutines return the complex hyperbolic tangent value.

Related Information

"catanh, catanhf, or catanhl Subroutine" on page 109

ctermid Subroutine

Purpose

Generates the path name of the controlling terminal.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdio.h>
char *ctermid ( String)
char *String;
```

Description

The ctermid subroutine generates the path name of the controlling terminal for the current process and stores it in a string.

Note: File access permissions depend on user access. Access to a file whose path name the ctermid subroutine has returned is not guaranteed.

The difference between the ctermid and ttyname subroutines is that the ttyname subroutine must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor. The ctermid subroutine returns a string (the /dev/tty file) that refers to the terminal if used as a file name. Thus, the **ttyname** subroutine is useful only if the process already has at least one file open to a terminal.

Parameters

String

If the String parameter is a null pointer, the string is stored in an internal static area and the address is returned. The next call to the ctermid subroutine overwrites the contents of the internal

If the String parameter is not a null pointer, it points to a character array of at least L ctermid elements as defined in the stdio.h file. The path name is placed in this array and the value of the String parameter is returned.

Related Information

The **isatty** or **ttyname** subroutine.

Input and Output Handling Programmer's Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine

Purpose

Converts the formats of date and time representations.

Library

Standard C Library (libc.a)

Syntax

```
#include <time.h>
char *ctime ( Clock)
const time_t *Clock;
struct tm *localtime (Clock)
const time_t *Clock;
struct tm *gmtime (Clock)
const time_t *Clock;
time t mktime( Timeptr)
struct tm *Timeptr;
double difftime( Time1, Time0)
time t Time0, Time1;
```

```
char *asctime ( Tm)
const struct tm *Tm;
void tzset ()
extern long int timezone;
extern int daylight;
extern char *tzname[];
```

Description

Attention: Do not use the tzset subroutine when linking with both libc.a and libbsd.a. The tzset subroutine sets the global external variable called **timezone**, which conflicts with the **timezone** subroutine in **libbsd.a**. This name collision may cause unpredictable results.

Attention: Do not use the ctime, localtime, gmtime, or asctime subroutine in a multithreaded environment. See the multithread alternatives in the **ctime r** ("ctime r, localtime r, gmtime r, or asctime r Subroutine" on page 166), localtime r, gmtime r, or asctime r subroutine article.

The **ctime** subroutine converts a time value pointed to by the *Clock* parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into a 26-character string in the following form:

```
Sun Sept 16 01:03:52 1973\n\0
```

The width of each field is always the same as shown here.

The **ctime** subroutine adjusts for the time zone and daylight saving time, if it is in effect.

The **localtime** subroutine converts the long integer pointed to by the *Clock* parameter, which contains the time in seconds since 00:00:00 UTC, 1 January 1970, into a tm structure. The localtime subroutine adjusts for the time zone and for daylight-saving time, if it is in effect. Use the time-zone information as though localtime called tzset.

The **gmtime** subroutine converts the long integer pointed to by the *Clock* parameter into a **tm** structure containing the Coordinated Universal Time (UTC), which is the time standard the operating system uses.

Note: UTC is the international time standard intended to replace GMT.

The **tm** structure is defined in the **time.h** file, and it contains the following members:

```
/* Seconds (0 - 59) */
int tm sec;
               /* Minutes (0 - 59) */
int tm min;
             /* Hours (0 - 23) */
int tm hour;
int tm mday;
             /* Day of month (1 - 31) */
int tm mon;
              /* Month of year (0 - 11) */
int tm year;
             /* Year - 1900 */
            /* Day of week (Sunday = 0) */
int tm_wday;
int tm yday;
             /* Day of year (0 - 365) */
int tm isdst; /* Nonzero = Daylight saving time */
```

The mktime subroutine is the reverse function of the localtime subroutine. The mktime subroutine converts the tm structure into the time in seconds since 00:00:00 UTC, 1 January 1970. The tm wday and tm yday fields are ignored, and the other components of the tm structure are not restricted to the ranges specified in the /usr/include/time.h file. The value of the tm isdst field determines the following actions of the **mktime** subroutine:

- 0 Initially presumes that Daylight Savings Time (DST) is not in effect.
- >0 Initially presumes that DST is in effect.
- -1 Actively determines whether DST is in effect from the specified time and the local time zone. Local time zone information is set by the tzset subroutine.

Upon successful completion, the mktime subroutine sets the values of the tm wday and tm yday fields appropriately. Other fields are set to represent the specified time since January 1, 1970. However, the values are forced to the ranges specified in the /usr/include/time.h file. The final value of the tm mday field is not set until the values of the tm mon and tm year fields are determined.

Note: The mktime subroutine cannot convert time values before 00:00:00 UTC, January 1, 1970 and after 03:14:07 UTC, January 19, 2038.

The difftime subroutine computes the difference between two calendar times: the Time1 and -Time0 parameters.

The **asctime** subroutine converts a **tm** structure to a 26-character string of the same format as **ctime**.

If the TZ environment variable is defined, then its value overrides the default time zone, which is the U.S. Eastern time zone. The environment facility contains the format of the time zone information specified by TZ. TZ is usually set when the system is started with the value that is defined in either the /etc/environment or /etc/profile files. However, it can also be set by the user as a regular environment variable for performing alternate time zone conversions.

The tzset subroutine sets the timezone, daylight, and tzname external variables to reflect the setting of TZ. The tzset subroutine is called by ctime and localtime, and it can also be called explicitly by an application program.

The timezone external variable contains the difference, in seconds, between UTC and local standard time. For example, the value of **timezone** is 5 * 60 * 60 for U.S. Eastern Standard Time.

The daylight external variable is nonzero when a daylight-saving time conversion should be applied. By default, this conversion follows the standard U.S. conventions; other conventions can be specified. The default conversion algorithm adjusts for the peculiarities of U.S. daylight saving time in 1974 and 1975.

The tzname external variable contains the name of the standard time zone (tzname[0]) and of the time zone when Daylight Savings Time is in effect (tzname[1]). For example:

```
char *tzname[2] = {"EST", "EDT"};
```

The **time.h** file contains declarations of all these subroutines and externals and the **tm** structure.

Parameters

Clock Specifies the pointer to the time value in seconds.

Timeptr Specifies the pointer to a tm structure. Time1 Specifies the pointer to a time_t structure. Time0 Specifies the pointer to a **time t** structure. Specifies the pointer to a tm structure. Tm

Return Values

Attention: The return values point to static data that is overwritten by each call.

The **tzset** subroutine returns no value.

The **mktime** subroutine returns the specified time in seconds encoded as a value of type **time_t**. If the time cannot be represented, the function returns the value (time_t)-1.

The **localtime** and **gmtime** subroutines return a pointer to the **struct tm**.

The **ctime** and **asctime** subroutines return a pointer to a 26-character string.

The difftime subroutine returns the difference expressed in seconds as a value of type double.

Related Information

The getenv ("getenv Subroutine" on page 313) subroutine, gettimer ("gettimer, settimer, restimer, stime, or time Subroutine" on page 375) subroutine, strftime subroutine.

List of Time Data Manipulation Services in AIX 5L Version 5.2 System Management Concepts: Operating System and Devices.

National Language Support Overview for Programming in AIX 5L Version 5.2 National Language Support Guide and Reference.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

ctime_r, localtime_r, gmtime_r, or asctime_r Subroutine

Purpose

Converts the formats of date and time representations.

Library

Thread-Safe C Library (libc_r.a)

Syntax

```
#include <time.h>
char *ctime r(Timer, BufferPointer)
const time t * Timer;
char * BufferPointer;
struct tm *localtime_r(Timer, CurrentTime)
const time t * Timer;
struct tm * CurrentTime;
struct tm *gmtime_r(Timer, XTime)
const time_t * Timer;
struct tm * XTime;
char *asctime r(TimePointer, BufferPointer)
const struct tm * TimePointer;
char * BufferPointer;
```

Description

The **ctime** r subroutine converts a time value pointed to by the *Timer* parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into the character array pointed to by the BufferPointer parameter. The character array should have a length of at least 26 characters so the converted time value fits without truncation. The converted time value string takes the form of the following example:

```
Sun Sep 16 01:03:52 1973\n\0
```

The width of each field is always the same as shown here.

The **ctime** r subroutine adjusts for the time zone and daylight saving time, if it is in effect.

The localtime_r subroutine converts the time_t structure pointed to by the Timer parameter, which contains the time in seconds since 00:00:00 UTC, January 1, 1970, into the tm structure pointed to by the CurrentTime parameter. The localtime r subroutine adjusts for the time zone and for daylight saving time, if it is in effect.

The **qmtime** r subroutine converts the **time** t structure pointed to by the *Timer* parameter into the tm structure pointed to by the XTime parameter.

The tm structure is defined in the time.h header file. The time.h file contains declarations of these subroutines, externals, and the tm structure.

The asctime_r subroutine converts the tm structure pointed to by the TimePointer parameter into a 26-character string in the same format as the ctime_r subroutine. The results are placed into the character array, BufferPointer. The BufferPointer parameter points to the resulting character array, which takes the form of the following example:

Sun Sep 16 01:03:52 1973\n\0

Programs using this subroutine must link to the libpthreads.a library.

Parameters

Timer Points to a time_t structure, which contains the number of seconds since 00:00:00 UTC,

January 1, 1970.

BufferPointer Points to a character array at least 26 characters long.

CurrentTime Points to a **tm** structure. The result of the **localtime_r** subroutine is placed here.

XTime Points to a **tm** structure used for the results of the **qmtime r** subroutine. Points to a tm structure used as input to the asctime_r subroutine. TimePointer

Return Values

The **localtime r** and **gmtime r** subroutines return a pointer to the **tm** structure. The **asctime r** returns NULL if either TimePointer or BufferPointer are NULL.

The ctime_r and asctime_r subroutines return a pointer to a 26-character string. The ctime_r subroutine returns NULL if the BufferPointer is NULL.

Files

/usr/include/time.h

Defines time macros, data types, and structures.

Related Information

The ctime, localtime, gmtime, mktime, difftime, asctime, or tzset ("ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine" on page 163) subroutine.

Subroutines, Example Programs, and Libraries and List of Multi-threaded Programming Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines

Purpose

Classifies characters.

Library

Standard Character Library (libc.a)

Syntax

```
#include <ctype.h>
int isalpha ( Character)
int Character;
int isupper (Character)
int Character;
int islower (Character)
int Character;
int isdigit (Character)
int Character;
int isxdigit (Character)
int Character;
int isalnum (Character)
int Character;
int isspace (Character)
int Character;
int ispunct (Character)
int Character;
int isprint (Character)
int Character;
int isgraph (Character)
int Character;
int iscntrl (Character)
int Character;
int isascii (Character)
int Character;
```

Description

The **ctype** subroutines classify character-coded integer values specified in a table. Each of these subroutines returns a nonzero value for True and 0 for False.

Note: The **ctype** subroutines should only be used on character data that can be represented by a single byte value (0 through 255). Attempting to use the **ctype** subroutines on multi-byte locale data may give inconsistent results. Wide character classification routines (such as **iswprint**, **iswlower**, etc.) should be used with dealing with multi-byte character data.

Locale Dependent Character Tests

The following subroutines return nonzero (True) based upon the character class definitions for the current locale.

isalnum

Returns nonzero for any character for which the **isalpha** or **isdigit** subroutine would return nonzero. The **isalnum** subroutine tests whether the character is of the **alpha** or **digit** class.

isalpha Returns nonzero for any character for which the isupper or islower subroutines would return

nonzero. The **isalpha** subroutine also returns nonzero for any character defined as an alphabetic character in the current locale, or for a character for which *none* of the **iscntrl**, **isdigit**, **ispunct**, or **isspace** subroutines would return nonzero. The **isalpha** subroutine tests whether the

character is of the alpha class.

isupper Returns nonzero for any uppercase letter [A through Z]. The **isupper** subroutine also returns

nonzero for any character defined to be uppercase in the current locale. The isupper subroutine

tests whether the character is of the upper class.

islower Returns nonzero for any lowercase letter [a through z]. The islower subroutine also returns

nonzero for any character defined to be lowercase in the current locale. The islower subroutine

tests whether the character is of the lower class.

isspace Returns nonzero for any white-space character (space, form feed, new line, carriage return,

horizontal tab or vertical tab). The isspace subroutine tests whether the character is of the

space class.

ispunct Returns nonzero for any character for which the **isprint** subroutine returns nonzero, except the

space character and any character for which the isalnum subroutine would return nonzero. The

ispunct subroutine also returns nonzero for any locale-defined character specified as a

punctuation character. The **ispunct** subroutine tests whether the character is of the **punct** class.

isprint Returns nonzero for any printing character. Returns nonzero for any locale-defined character that

is designated a printing character. This routine tests whether the character is of the **print** class.

isgraph Returns nonzero for any character for which the **isprint** character returns nonzero, except the

space character. The **isgraph** subroutine tests whether the character is of the **graph** class.

Returns nonzero for any character for which the **isprint** subroutine returns a value of False (0)

and any character that is designated a control character in the current locale. For the C locale, control characters are the ASCII delete character (0177 or 0x7F), or an ordinary control character (less than 040 or 0x20). The **iscntrl** subroutine tests whether the character is of the **cntrl** class.

Locale Independent Character Tests

The following subroutines return nonzero for the same characters, regardless of the locale:

isdigit Character is a digit in the range [0 through 9].

isxdigit Character is a hexadecimal digit in the range [0 through 9], [A through F], or [a through f].
isascii Character is an ASCII character whose value is in the range 0 through 0177 (0 through 0x7F),

inclusive.

Parameter

iscntrl

Character Indicates the character to be tested (integer value).

Return Codes

The **ctype** subroutines return nonzero (True) if the character specified by the *Character* parameter is a member of the selected character class; otherwise, a 0 (False) is returned.

Related Information

The setlocale subroutine.

List of Character Manipulation Services and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

cuserid Subroutine

Purpose

Gets the alphanumeric user name associated with the current process.

Library

Standard C Library (libc.a)

Use the **libc r.a** library to access the thread-safe version of this subroutine.

Syntax

```
#include <stdio.h>
char *cuserid ( Name)
char *Name;
```

Description

The **cuserid** subroutine gets the alphanumeric user name associated with the current process. This subroutine generates a character string representing the name of a process's owner.

Note: The cuserid subroutine duplicates functionality available with the getpwuid and getuid subroutines. Present applications should use the getpwuid and getuid subroutines.

If the Name parameter is a null pointer, then a character string of size L cuserid is dynamically allocated with malloc, and the character string representing the name of the process owner is stored in this area. The cuserid subroutine then returns the address of this area. Multithreaded application programs should use this functionality to obtain thread specific data, and then continue to use this pointer in subsequent calls to the **curserid** subroutine. In any case, the application program must deallocate any dynamically allocated space with the **free** subroutine when the data is no longer needed.

If the Name parameter is not a null pointer, the character string is stored into the array pointed to by the Name parameter. This array must contain at least the number of characters specified by the constant L cuserid. This constant is defined in the stdio.h file.

If the user name cannot be found, the **cuserid** subroutine returns a null pointer; if the Name parameter is not a null pointer, a null character ('\0') is stored in Name [0].

Parameter

Name Points to a character string representing a user name.

Related Information

The endpwent ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 354) subroutine, getlogin ("getlogin Subroutine" on page 335), getpwent ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 354), getpwnam ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 354), getpwuid ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 354), or putpwent ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 354) subroutine.

Input and Output Handling Programmer's Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

defssys Subroutine

Purpose

Initializes the SRCsubsys structure with default values.

Library

System Resource Controller Library (libsrc.a)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
void defssys( SRCSubsystem)
struct SRCsubsys *SRCSubsystem;
```

Description

The defssys subroutine initializes the SRCsubsys structure of the /usr/include/sys/srcobj.h file with the following default values:

Field	Value
display	SRCYES
multi	SRCNO
contact	SRCSOCKET
waittime	TIMELIMIT
priority	20
action	ONCE
standerr	/dev/console
standin	/dev/console
standout	/dev/console

All other numeric fields are set to 0, and all other alphabetic fields are set to an empty string.

This function must be called to initialize the SRCsubsys structure before an application program uses this structure to add records to the subsystem object class.

Parameters

SRCSubsystem Points to the SRCsubsys structure.

Related Information

The **addssys** ("addssys Subroutine" on page 19) subroutine.

Defining Your Subsystem to the SRC, List of SRC Subroutines, System Resource Controller (SRC) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

delssys Subroutine

Purpose

Removes the subsystem objects associated with the SubsystemName parameter.

Library

System Resource Controller Library (libsrc.a)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
int delssys ( SubsystemName)
char *SubsystemName;
```

Description

The **delssys** subroutine removes the subsystem objects associated with the specified subsystem. This removes all objects associated with that subsystem from the following object classes:

- Subsystem
- · Subserver Type
- Notify

The program running with this subroutine must be running with the group system.

Parameter

SubsystemName Specifies the name of the subsystem.

Return Values

Upon successful completion, the delssys subroutine returns a positive value. If no record is found, a value of 0 is returned. Otherwise, -1 is returned and the odmerrno variable is set to indicate the error. See "Appendix B. ODM Error Codes (Appendix B, "ODM Error Codes," on page 1003)" for a description of possible odmerrno values.

Security

Privilege Control:

SET_PROC_AUDIT kernel privilege

Files Accessed:

Mode File

/etc/objrepos/SRCsubsys 644 644 /etc/objrepos/SRCsubsvr 644 /etc/objrepos/SRCnotify

Auditing Events:

Information **Event**

SRC_Delssys Lists in an audit log the name of the subsystem being removed.

Files

/etc/objrepos/SRCsubsys SRC Subsystem Configuration object class. /etc/objrepos/SRCsubsvr SRC Subsystem Configuration object class.

/etc/objrepos/SRCnotify /dev/SRC /dev/.SRC-unix /usr/include/sys/srcobj.h /usr/include/spc.h

SRC Notify Method object class. Specifies the AF_UNIX socket file. Specifies the location for temporary socket files. Defines object structures used by the SRC. Defines external interfaces provided by the SRC subroutines.

Related Information

The addssys ("addssys Subroutine" on page 19) subroutine, chssys ("chssys Subroutine" on page 132) subroutine.

The chssys command, mkssys command, rmssys command.

List of SRC Subroutines and System Resource Controller (SRC) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

dirname Subroutine

Purpose

Report the parent directory name of a file path name.

Library

Standard C Library (libc.a)

Syntax

#include <libgen.h>

char *dirname (path) char *path

Description

Given a pointer to a character string that contains a file system path name, the dirname subroutine returns a pointer to a string that is the parent directory of that file. Trailing "/" characters in the path are not counted as part of the path.

If path is a null pointer or points to an empty string, a pointer to a static constant "." is returned.

The **dirname** and **basename** subroutines together yield a complete path name. **dirname** (path) is the directory where **basename** (path) is found.

Parameters

Character string containing a file system path name. path

Return Values

The **dirname** subroutine returns a pointer to a string that is the parent directory of path. If path or *path is a null pointer or points to an empty string, a pointer to a string "." is returned. The dirname subroutine may modify the string pointed to by path and may return a pointer to static storage that may then be overwritten by sequent calls to the **dirname** subroutine.

Examples

A simple file name and the strings "." and ".." all have "." as their return value.

Input string	Output string
/usr/lib	/usr
/usr/	/
usr	
/	/
	•

The following code reads a path name, changes directory to the appropriate directory, and opens the file.

```
char path [MAXPATHEN], *pathcopy;
int fd;
fgets (path, MAXPATHEN, stdin);
pathcopy = strdup (path);
chdir (dirname (pathcopy) );
fd = open (basename (path), O_RDONLY);
```

Related Information

The basename ("basename Subroutine" on page 96) or chdir ("chdir Subroutine" on page 123) subroutine.

disclaim Subroutine

Purpose

Disclaims the content of a memory address range.

Syntax

```
#include <sys/shm.h>
int disclaim ( Address, Length, Flag)
char *Address;
unsigned int Length, Flag;
```

Description

The disclaim subroutine marks an area of memory having content that is no longer needed. The system then stops paging the memory area. This subroutine cannot be used on memory that is mapped to a file by the shmat subroutine.

Parameters

Address Points to the beginning of the memory area. Length Specifies the length of the memory area in bytes.

Flag Must be the value ZERO_MEM, which indicates that each memory location in the address range

should be set to 0.

Return Values

When successful, the **disclaim** subroutine returns a value of 0.

Error Codes

If the disclaim subroutine is unsuccessful, it returns a value of -1 and sets the errno global variable to indicate the error. The disclaim subroutine is unsuccessful if one or more of the following are true:

EFAULT The calling process does not have write access to the area of memory that begins at the

Address parameter and extends for the number of bytes specified by the Length parameter.

EINVAL The value of the *Flag* parameter is not valid.

EINVAL The memory area is mapped to a file.

dladdr Subroutine

Purpose

Translates address to symbolic information.

Syntax

```
#include <dlfcn.h>
int dladdr(void *address, Dl info *dlip);
```

Description

dladdr allows a process to obtain information about the symbol that most closely defines a given address. dladdr first determines whether the specified address is located within one of the mapped objects (executable or shared objects) that make up the process' address space. An address is deemed to fall within a mapped object when it is between the base address at which the object was mapped and the highest virtual address mapped for that object, inclusive. If a mapped object fits this criteria, its dynamic symbol table is searched to locate the nearest symbol to the specified address. The nearest symbol is the one whose value is equal to, or closest to but less than, the specified address.

dlip is a pointer to a **DI info** structure. The structure must be allocated by the user. The structure members are set by dladdr if the specified address falls within one of the mapped objects. The Dl_info structure contains at least the following members:

```
const char
              *dli fname;
             *dli_fbase;
void
const char *dli_sname;
void
              *dli saddr;
size t dli size;
              dli bind;
int
int
              dli_type;
```

Descriptions of these members appear below:

dli fname

Contains a pointer to the filename of the mapped object containing address.

dli fbase

Contains the base address of the mapped object containing address.

dli_sname

Contains a pointer to the name of the nearest symbol to the specified address. This symbol either has the same address, or is the nearest symbol with a lower address.

dli saddr

Contains the actual address of the nearest symbol.

dli_size

Contains the size of the nearest symbol as defined in the dynamic symbol table.

If no symbol is found within the object containing address whose value is less than or equal to address, the dli sname, dli saddr and dli size members are set to 0; the dli bind member is set to STB_LOCAL, and the dli_type member is set to STT_NOTYPE.

For the a.out, the symbol table created by Id for use by the dynamic linker might contain only a subset of the symbols defined in the a.out [see dlopen ("dlopen Subroutine" on page 178)]. This could cause dladdr to return information for a symbol that is actually unrelated to the specified address.

The addresses and the strings pointed to by the members of the **DI_info** structure refer to locations within mapped objects. These may become invalid if the objects are unmapped from the address space [see diclose ("dlclose Subroutine")]

Return values

If the specified address does not fall within one of the mapped objects, 0 is returned; the contents of the DI_info structure are unspecified. Otherwise, a non-zero value is returned and the associated DI_info elements are set.

Related Information

The diclose ("dlclose Subroutine") subroutine, dierror ("dlerror Subroutine" on page 177) subroutine, dlopen ("dlopen Subroutine" on page 178) subroutine, and dlsym ("dladdr Subroutine" on page 175) subroutine.

dlclose Subroutine

Purpose

Closes and unloads a module loaded by the **dlopen** subroutine.

Syntax

#include <dlfcn.h> int dlclose(Data); void *Data;

Description

The diclose subroutine is used to remove access to a module loaded with the dlopen subroutine. In addition, access to dependent modules of the module being unloaded is removed as well.

Modules being unloaded with the dlclose subroutine will not be removed from the process's address space if they are still required by other modules. Nevertheless, subsequent uses of Data are invalid, and further uses of symbols that were exported by the module being unloaded result in undefined behavior.

Parameters

Data A loaded module reference returned from a previous call to dlopen.

Return Values

Upon successful completion, 0 (zero) is returned. Otherwise, errno is set to EINVAL, and the return value is also EINVAL. Even if the diclose subroutine succeeds, the specified module may still be part of the process's address space if the module is still needed by other modules.

Error Codes

EINVAL

The Data parameter does not refer to a module opened by dlopen that is still open. The parameter may be corrupt or the module may have been unloaded by a previous call to diclose.

Related Information

The **dierror** ("dierror Subroutine") subroutine, **diopen** ("diopen Subroutine" on page 178) subroutine, dlsym ("dladdr Subroutine" on page 175) subroutine, load ("load Subroutine" on page 618) subroutine, loadquery ("loadquery Subroutine" on page 623) subroutine, unload subroutine, loadbind ("loadbind Subroutine" on page 622) subroutine.

The Id command.

The Shared Libraries and Shared Memory Overview and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

dlerror Subroutine

Purpose

Returns a pointer to information about the last dlopen, dlsym, or dlclose error.

Syntax

#include <dlfcn.h> char *dlerror(void);

Description

The dierror subroutine is used to obtain information about the last error that occurred in a dynamic loading routine (that is, dlopen, dlsym, or dlclose). The returned value is a pointer to a null-terminated string without a final newline. Once a call is made to this function, subsequent calls without any intervening dynamic loading errors will return NULL.

Applications can avoid calling the dlerror subroutine, in many cases, by examining errno after a failed call to a dynamic loading routine. If errno is ENOEXEC, the dlerror subroutine will return additional information. In all other cases, **dlerror** will return the string corresponding to the value of **errno**.

The **dierror** function may invoke **loadquery** to ascertain reasons for a failure. If a call is made to **load** or unload between calls to dlopen and dlerror, incorrect information may be returned.

Return Values

A pointer to a static buffer is returned; a NULL value is returned if there has been no error since the last call to dlerror. Applications should not write to this buffer; they should make a copy of the buffer if they wish to preserve the buffer's contents.

Related Information

The load ("load Subroutine" on page 618) subroutine, loadbind ("loadbind Subroutine" on page 622) subroutine, loadquery ("loadquery Subroutine" on page 623)subroutine, unload subroutine, dlopen ("dlopen Subroutine" on page 178) subroutine, dlclose ("dlclose Subroutine" on page 176) subroutine, dlsym ("dladdr Subroutine" on page 175) subroutine.

The Id command.

The Shared Libraries and Shared Memory Overview and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

dlopen Subroutine

Purpose

Dynamically load a module into the calling process.

Syntax

#include <dlfcn.h> void *dlopen (FilePath, Flags); const char *FilePath; int Flags;

Description

The **dlopen** subroutine loads the module specified by *FilePath* into the executing process's address space. Dependents of the module are automatically loaded as well. If the module is already loaded, it is not loaded again, but a new, unique value will be returned by the dlopen subroutine.

The value returned by **dlopen** may be used in subsequent calls to **dlsym** and **dlclose**. If an error occurs during the operation, dlopen returns NULL.

If the main application was linked with the -brtl option, then the runtime linker is invoked by dlopen. If the module being loaded was linked with runtime linking enabled, both intra-module and inter-module references are overridden by any symbols available in the main application. If runtime linking was enabled, but the module was not built enabled, then all inter-module references will be overridden, but some intra-module references will not be overridden.

If the module being opened with **dlopen** or any of its dependents is being loaded for the first time, initialization routines for these newly-loaded routines are called (after runtime linking, if applicable) before dlopen returns. Initialization routines are the functions specified with the -binitfini: linker option when the module was built. (Refer to the **Id** command for more information about this option.)

Notes:

- 1. The initialization functions need not have any special names, and multiple functions per module are allowed.
- 2. If the module being loaded has read-other permission, the module is loaded into the global shared library segment. Modules loaded into the global shared library segment are not unloaded even if they are no longer being used. Use the slibclean command to remove unused modules from the global shared library segment.

Use the environment variable LIBPATH to specify a list of directories in which **dlopen** search es for the named module. The running application also contains a set of library search paths that were specified when the application was linked; these paths are searched after any paths found in LIBPATH. Also, the setenv subroutine

FilePath

Specifies the name of a file containing the loadable module. This parameter can be contain an absolute path, a relative path, or no path component. If FilePath contains a slash character, FilePath is used directly, and no directories are searched.

If the FilePath parameter is /unix, dlopen returns a value that can be used to look up symbols in the current kernel image, including those symbols found in any kernel extension that was available at the time the process began execution.

If the value of *FilePath* is NULL, a value for the main application is returned. This allows dynamically loaded objects to look up symbols in the main executable, or for an application to examine symbols available within itself.

Flags

Specifies variations of the behavior of dlopen. Either RTLD NOW or RTLD LAZY must always be specified. Other flags may be OR'ed with RTLD NOW or RTLD LAZY.

RTLD_NOW Load all dependents of the module being loaded and resolve all symbols.

Specifies the same behavior as RTLD_NOW. In a future release of the operating RTLD_LAZY

system, the behavior of the RTLD_LAZY may change so that loading of dependent

modules is deferred of resolution of some symbols is deferred.

RTLD_GLOBAL Allows symbols in the module being loaded to be visible when resolving symbols

used by other **dlopen** calls. These symbols will also be visible when the main

application is opened with dlopen(NULL, mode).

Prevent symbols in the module being loaded from being used when resolving RTLD_LOCAL

symbols used by other dlopen calls. Symbols in the module being loaded can only

be accessed by calling dlsym subroutine. If neither RTLD_GLOBAL nor

RTLD_LOCAL is specified, the default is RTLD_LOCAL. If both flags are specified,

RTLD_LOCAL is ignored.

The **dlopen** subroutine can be used to load a module that is a member of an archive. RTLD_MEMBER

> The L LOADMEMBER flag is used when the load subroutine is called. The module name FilePath names the archive and archive member according to the rules outlined

in the load subroutine.

RTLD_NOAUTODEFER Prevents deferred imports in the module being loaded from being automatically

resolved by subsequent loads. The L_NOAUTODEFER flag is used when the load

subroutine is called.

Ordinarily, modules built for use by the **dlopen** and **dlsym** sub routines will not contain deferred imports. However, deferred imports can be still used. A module opened with **dlopen** may provide definitions for deferred imports in the main application, for modules loaded with the load subroutine (if the L_NOAUTODEFER flag was not used), and for other modules loaded with the dlopen subroutine (if the

RTLD_NOAUTODEFER flag was not used).

Return Values

Upon successful completion, dlopen returns a value that can be used in calls to the dlsym and dlclose subroutines. The value is not valid for use with the **loadbind** and **unload** subroutines.

If the dlopen call fails, NULL (a value of 0) is returned and the global variable errno is set. If errno contains the value ENOEXEC, further information is available via the dlerror function.

Error Codes

See the **load** subroutine for a list of possible **errno** values and their meanings.

Related Information

The **diclose** ("dlclose Subroutine" on page 176) subroutine, **dlerror** ("dlerror Subroutine" on page 177) subroutine, **dlsym** ("dladdr Subroutine" on page 175) subroutine, **load** ("load Subroutine" on page 618) subroutine, **loadbind** ("loadbind Subroutine" on page 622) subroutine, **loadquery** ("loadquery Subroutine" on page 623)subroutine, **unload** subroutine.

The **Id** command.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

dlsym Subroutine

Purpose

Looks up the location of a symbol in a module that is loaded with **dlsym**.

Syntax

```
#include <dlfcn.h>
void *dlsym(Data, Symbol);
void *Data;
const char *Symbol;
```

Description

The **disym** subroutine looks up a named symbol exported from a module loaded by a previous call to the **diopen** subroutine. Only exported symbols are found by **disym**. See the **Id** command to see how to export symbols from a module.

Data Specifies a value returned by a previous call to **dlopen**.

Symbol Specifies the name of a symbol exported from the referenced module. The form should be a

NULL-terminated string.

Note: C++ symbol names should be passed to **dlsym** in mangled form; **dlsym** does not perform any name demangling on behalf of the calling application.

A search for the named symbol is based upon breadth-first ordering of the module and its dependants. If the module was constructed using the **-G** or **-brtl** linker option, the module's dependants will include all modules named on the **Id** command line, in the original order. The dependants of a module that was not linked with the **-G** or **-brtl** linker option will be listed in an unspecified order.

Return Values

If the named symbol is found, its address is returned. If the named symbol is not found, NULL is returned and **errno** is set to 0. If *Data* or *Symbol* are invalid, NULL is returned and **errno** is set to **EINVAL**.

If the first definition found is an export of an imported symbol, this definition will satisfy the search. The address of the imported symbol is returned. If the first definition is a deferred import, the definition is ignored and the search continues.

If the named symbol refers to a BSS symbol (uninitialized data structure), the search continues until an initialized instance of the symbol is found or the module and all of its dependants have been searched. If an initialized instance is found, its address is returned; otherwise, the address of the first uninitialized instance is returned.

Error Codes

EINVAL

If the Data parameter does not refer to a module opened by dlopen that is still loaded or if the Symbol parameter points to an invalid address, the dlsym subroutine returns NULL and errno is set to EINVAL.

Related Information

The diclose ("dlclose Subroutine" on page 176) subroutine, dlerror ("dlerror Subroutine" on page 177) subroutine, **dlopen** ("dlopen Subroutine" on page 178) subroutine, **load** ("load Subroutine" on page 618) subroutine, loadbind ("loadbind Subroutine" on page 622) subroutine, loadquery ("loadquery Subroutine" on page 623) subroutine, unload subroutine.

The Id command.

drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, or srand48 Subroutine

Purpose

Generate uniformly distributed pseudo-random number sequences.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
double drand48 (void)
double erand48 ( xsubi)
unsigned short int xsubi[3];
long int jrand48 (xsubi)
unsigned short int xsubi[3];
void lcong48 ( Parameter)
unsigned short int Parameter[7];
long int lrand48 (void)
long int mrand48 (void)
long int nrand48 (xsubi)
unsigned short int xsubi[3];
unsigned short int *seed48 ( Seed16v)
unsigned short int Seed16v[3];
void srand48 ( SeedValue)
long int SeedValue;
```

Description

Attention: Do not use the drand48, erand48, irand48, Icong48, Irand48, mrand48, nrand48, seed48, or srand48 subroutine in a multithreaded environment.

This family of subroutines generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic.

The drand48 subroutine and the erand48 subroutine return positive double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

The Irand48 subroutine and the nrand48 subroutine return positive long integers uniformly distributed over the interval [0,2**31).

The mrand48 subroutine and the jrand48 subroutine return signed long integers uniformly distributed over the interval [-2**31, 2**31).

The **srand48** subroutine, **seed48** subroutine, and **Icong48** subroutine initialize the random-number generator. Programs must call one of them before calling the drand48, Irand48 or mrand48 subroutines. (Although it is not recommended, constant default initializer values are supplied if the drand48, Irand48 or mrand48 subroutines are called without first calling an initialization subroutine.) The erand48, nrand48, and jrand48 subroutines do not require that an initialization subroutine be called first.

The previous value pointed to by the **seed48** subroutine is stored in a 48-bit internal buffer, and a pointer to the buffer is returned by the **seed48** subroutine. This pointer can be ignored if it is not needed, or it can be used to allow a program to restart from a given point at a later time. In this case, the pointer is accessed to retrieve and store the last value pointed to by the seed48 subroutine, and this value is then used to reinitialize, by means of the seed48 subroutine, when the program is restarted.

All the subroutines work by generating a sequence of 48-bit integer values, x[i], according to the linear congruential formula:

```
x[n+1] = (ax[n] + c) \mod m, n is > = 0
```

The parameter m = 248; hence 48-bit integer arithmetic is performed. Unless the **Icong48** subroutine has been called, the multiplier value a and the addend value c are:

```
a = 5DEECE66D base 16 = 273673163155 base 8
c = B base 16 = 13 base 8
```

Parameters

xsubi

Specifies an array of three shorts, which, when concatenated together, form a 48-bit integer. SeedValue Specifies the initialization value to begin randomization. Changing this value changes the randomization pattern. Seed16v Specifies another seed value; an array of three unsigned shorts that form a 48-bit seed value. Parameter Specifies an array of seven shorts, which specifies the initial xsubi value, the multiplier value a and

the add-in value c.

Return Values

The value returned by the drand48, erand48, jrand48, lrand48, nrand48, and mrand48 subroutines is computed by first generating the next 48-bit x[i] in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (most significant) bits of x[i] and transformed into the returned value.

The **drand48**, **Irand48**, and **mrand48** subroutines store the last 48-bit x[i] generated into an internal buffer; this is why they must be initialized prior to being invoked.

The erand48, irand48, and nrand48 subroutines require the calling program to provide storage for the successive x[i] values in the array pointed to by the xsubi parameter. This is why these routines do not have to be initialized; the calling program places the desired initial value of x[i] into the array and pass it as a parameter.

By using different parameters, the erand48, jrand48, and nrand48 subroutines allow separate modules of a large program to generate independent sequences of pseudo-random numbers. In other words, the sequence of numbers that one module generates does not depend upon how many times the subroutines are called by other modules.

The **Iconq48** subroutine specifies the initial x[i] value, the multiplier value a, and the addend value c. The Parameter array elements Parameter[0-2] specify x[i], Parameter[3-5] specify the multiplier a, and Parameter [6] specifies the 16-bit addend c. After Iconq48 has been called, a subsequent call to either the **srand48** or **seed48** subroutine restores the standard *a* and *c* specified before.

The initializer subroutine **seed48** sets the value of x[i] to the 48-bit value specified in the array pointed to by the Seed16v parameter. In addition, seed48 returns a pointer to a 48-bit internal buffer that contains the previous value of x[i] that is used only by **seed48**. The returned pointer allows you to restart the pseudo-random sequence at a given point. Use the pointer to copy the previous x[i] value into a temporary array. Then call seed48 with a pointer to this array to resume processing where the original sequence stopped.

The initializer subroutine **srand48** sets the high-order 32 bits of x[i] to the 32 bits contained in its parameter. The low order 16 bits of x[i] are set to the arbitrary value 330E16.

Related Information

The rand, srand subroutine, random, srandom, initstate, or setstate subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

drem Subroutine

Purpose

Computes the IEEE Remainder as defined in the IEEE Floating-Point Standard.

Libraries

IEEE Math Library (libm.a) or System V Math Library (libmsaa.a)

Syntax

```
#include <math.h>
double drem (x, y)
double x, y;
```

Description

The **drem** subroutine calculates the remainder r equal to x minus n to the x power multiplied by y (r = x - 1)n * y), where the n parameter is the integer nearest the exact value of x divided by y (x/y). If |n - x/y| =1/2, then the *n* parameter is an even value. Therefore, the remainder is computed exactly, and the absolute value of r(|r|) is less than or equal to the absolute value of y divided by 2(|y|/2).

The IEEE Remainder differs from the **fmod** subroutine in that the IEEE Remainder always returns an r parameter such that |r| is less than or equal to |y|/2, while FMOD returns an r such that |r| is less than or equal to |y|. The IEEE Remainder is useful for argument reduction for transcendental functions.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **-Im** flag. For example: compile the drem.c file:

```
cc drem.c -lm
```

Note: For new development, the remainder subroutine is the preferred interface.

Parameters

- x Specifies double-precision floating-point value.
- y Specifies a double-precision floating-point value.

Return Values

The **drem** subroutine returns a NaNQ value for (x, 0) and (+/-INF, y).

Related Information

The **floor**, **ceil**, **nearest**, **trunc**, **rint**, **itrunc**, **fmod**, **fabs**, or **uitruns** ("floor, floorf, floorl, nearest, trunc, itrunc, or uitrunc Subroutine" on page 234) subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

_end, _etext, or _edata Identifier

Purpose

Define the first addresses following the program, initialized data, and all data.

Syntax

```
extern _end;
extern _etext;
extern _edata;
```

Description

The external names **_end**, **_etext**, and **_edata** are defined by the loader for all programs. They are not subroutines but identifiers associated with the following addresses:

_etext The first address following the program text.

_edata The first address following the initialized data region.

_end The first address following the data region that is not initialized. The name end (with no

underscore) defines the same address as does _end (with underscore).

The break value of the program is the first location beyond the data. When a program begins running, this location coincides with **end**. However, many factors can change the break value, including:

- · The brk or sbrk subroutine
- · The malloc subroutine
- · The standard I/O subroutines
- The -p flag with the cc command

Therefore, use the **brk** or **sbrk(0)** subroutine, not the **end** address, to determine the break value of the program.

Related Information

The brk or sbrk ("brk or sbrk Subroutine" on page 100) subroutine, malloc ("malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, or valloc Subroutine" on page 657) subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

ecvt, fcvt, or gcvt Subroutine

Purpose

Converts a floating-point number to a string.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
char *ecvt ( Value, NumberOfDigits, DecimalPointer, Sign;)
double Value;
int NumberOfDigits, *DecimalPointer, *Sign;
char *fcvt (Value, NumberOfDigits, DecimalPointer, Sign;)
double Value;
int NumberOfDigits, *DecimalPointer, *Sign;
char *gcvt (Value, NumberOfDigits, Buffer;)
double Value;
int NumberOfDigits;
char *Buffer;
```

Description

The ecvt, fcvt, and gcvt subroutines convert floating-point numbers to strings.

The **ecvt** subroutine converts the *Value* parameter to a null-terminated string and returns a pointer to it. The NumberOfDigits parameter specifies the number of digits in the string. The low-order digit is rounded according to the current rounding mode. The ecvt subroutine sets the integer pointed to by the DecimalPointer parameter to the position of the decimal point relative to the beginning of the string. (A negative number means the decimal point is to the left of the digits given in the string.) The decimal point itself is not included in the string. The ecvt subroutine also sets the integer pointed to by the Sign parameter to a nonzero value if the Value parameter is negative and sets a value of 0 otherwise.

The fcvt subroutine operates identically to the ecvt subroutine, except that the correct digit is rounded for C or FORTRAN F-format output of the number of digits specified by the *NumberOfDigits* parameter.

Note: In the F-format, the NumberOfDigits parameter is the number of digits desired after the decimal point. Large numbers produce a long string of digits before the decimal point, and then NumberOfDigits digits after the decimal point. Generally, the gcvt and ecvt subroutines are more useful for large numbers.

The **gcvt** subroutine converts the *Value* parameter to a null-terminated string, stores it in the array pointed to by the Buffer parameter, and then returns the Buffer parameter. The qcvt subroutine attempts to produce a string of the NumberOfDigits parameter significant digits in FORTRAN F-format. If this is not possible, the E-format is used. The gcvt subroutine suppresses trailing zeros. The string is ready for

printing, complete with minus sign, decimal point, or exponent, as appropriate. The radix character is determined by the current locale (see **setlocale** subroutine). If the **setlocale** subroutine has not been called successfully, the default locale, POSIX, is used. The default locale specifies a . (period) as the radix character. The **LC_NUMERIC** category determines the value of the radix character within the current locale.

The **ecvt**, **fcvt**, and **gcvt** subroutines represent the following special values that are specified in ANSI/IEEE standards 754-1985 and 854-1987 for floating-point arithmetic:

Quiet NaN Indicates a quiet not-a-number (NaNQ)

Signalling NaN Indicates a signaling NaNS Infinity Indicates a INF value

The sign associated with each of these values is stored in the Sign parameter.

Note: A value of 0 can be positive or negative. In the IEEE floating-point, zeros also have signs and set the *Sign* parameter appropriately.

Attention: All three subroutines store the strings in a static area of memory whose contents are overwritten each time one of the subroutines is called.

Parameters

Value Specifies some double-precision floating-point value.

NumberOfDigits Specifies the number of digits in the string.

DecimalPointer Specifies the position of the decimal point relative to the beginning of the string.

Sign Specifies that the sign associated with the return value is placed in the Sign parameter. In

IEEE floating-point, since 0 can be signed, the Sign parameter is set appropriately for

signed 0.

Buffer Specifies a character array for the string.

Related Information

The atof, strtod, atoff, or strtof ("atof atoff Subroutine" on page 77) subroutine, fp_read_rnd, or fp_swap_rnd ("fp_read_rnd or fp_swap_rnd Subroutine" on page 259) subroutine, printf ("printf, fprintf, sprintf, sprintf, wsprintf, vprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine, scanf subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

EnableCriticalSections, BeginCriticalSection, and EndCriticalSection Subroutine

Purpose

Enables a thread to be exempted from timeslicing and signal suspension, and protects critical sections.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/thread_ctl.h>
int EnableCriticalSections(void);
void BeginCriticalSection(void);
void EndCriticalSection(void);
```

Description

When called, the **EnableCriticalSections** subroutine enables the thread to be exempted from timeslicing and signal suspension. Once that is done, the thread can call the BeginCriticalSection and EndCriticalSection subroutines to protect critical sections. Calling the BeginCriticalSection and EndCriticalSection subroutines with exemption disabled has no effect. The subroutines are safe for use by multithreaded applications.

Once the service is enabled, the thread can protect critical sections by calling the BeginCriticalSection and EndCriticalSection subroutines. Calling the BeginCriticalSection subroutine will exempt the thread from timeslicing and suspension. Calling the EndCriticalSection subroutine will clear exemption for the thread.

The **BeginCriticalSection** subroutine will not make a system call. The **EndCriticalSection** subroutine might make a system call if the thread was granted a benefit during the critical section. The purpose of the system call would be to notify the kernel that any posted but undelivered stop signals can be delivered, and any postponed timeslice can now be completed.

Return Values

The **EnableCriticalSections** subroutine returns a zero.

erf, erff, or erfl Subroutine

Purpose

Computes the error and complementary error functions.

Libraries

```
IEEE Math Library (libm.a)
or System V Math Library (libmsaa.a)
```

Syntax

```
#include <math.h>
double erf (x)
double x;
float erff (x)
float x;
long double erfl (x)
long double x;
```

Description

The erf, erff, and erfl subroutines return the error function of the x parameter, defined for the erf subroutine as the following:

```
erf(x) = (2/sqrt(pi) * (integral [0 to x] of exp(-(t**2)) dt)
erfc(x) = 1.0 - erf(x)
```

Note: Compile any routine that uses subroutines from the libm.a library with the -Im flag. To compile the erf.c file, for example, enter:

```
cc erf.c -lm
```

An application wishing to check for error situations should set errno to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these functions. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies a double-precision floating-point value.

Return Values

Upon successful completion, the erf, erff, and erfl subroutines return the value of the error function.

If x is NaN, a NaN is returned.

If x is ± 0 , ± 0 is returned.

If x is $\pm \ln f$, ± 1 is returned.

If x is subnormal, a range error may occur, and 2 * x/sqrt(pi) should be returned.

Related Information

"erfc, erfcf, or erfcl Subroutine," "exp, expf, or expl Subroutine" on page 205, "feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, _class, finite, isnan, or unordered Subroutines" on page 138.

The sqrt, sqrtf, or sqrtl Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

128-Bit long double Floating-Point Format in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

math.h in AIX 5L Version 5.2 Files Reference.

erfc, erfcf, or erfcl Subroutine

Purpose

Computes the complementary error function.

Syntax

```
#include <math.h>
float erfcf (x)
float x;
long double erfcl (x)
```

```
long double x;
double erfc (x)
double x;
```

Description

The **erfcf**, **erfcl**, and **erfc** subroutines compute the complementary error function 1.0 - **erf**(x).

An application wishing to check for error situations should set errno to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these functions. Upon return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the erfcf, erfcl, and erfc subroutines return the value of the complementary error function.

If the correct value would cause underflow and is not representable, a range error may occur. Either 0.0 (if representable), or an implementation-defined value is returned.

If x is NaN. a NaN is returned.

If x is ± 0 , ± 1 is returned.

If x is -Inf, +2 is returned.

If x is +Inf, +0 is returned.

If the correct value would cause underflow and is representable, a range error may occur and the correct value is returned.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, class, finite, isnan, or unordered Subroutines" on page 138.

math.h in AIX 5L Version 5.2 Files Reference.

errlog Subroutine

Purpose

Logs an application error to the system error log.

Library

Run-Time Services Library (librts.a)

Syntax

```
#include <sys/errids.h>
int errlog ( ErrorStructure, Length)
void *ErrorStructure;
unsigned int Length;
```

Description

The **errlog** subroutine writes an error log entry to the **/dev/error** file. The **errlog** subroutine is used by application programs.

The transfer from the **err_rec** structure to the error log is by a **write** subroutine to the **/dev/error** special file.

The **errdemon** process reads from the **/dev/error** file and writes the error log entry to the system error log. The timestamp, machine ID, node ID, and Software Vital Product Data associated with the resource name (if any) are added to the entry before going to the log.

Parameters

ErrorStructure

Points to an error record structure containing an error record. Valid error record structures are typed in the /usr/include/sys/err_rec.h file. The two error record structures available are err_rec and err_rec0. The err_rec structure is used when the detail_data field is required. When the detail_data field is not required, the err_rec0 structure is used.

```
struct err_rec0 {
   unsigned int error_id;
   char resource_name[ERR_NAMESIZE];
};
struct err_rec {
   unsigned int error_id;
   char resource_name[ERR_NAMESIZE];
   char detail_data[1];
};
```

The fields of the structures err_rec and err_rec0 are:

```
error_id
```

Specifies an index for the system error template database, and is assigned by the **errupdate** command when adding an error template. Use the **errupdate** command with the **-h** flag to get a #define statement for this 8-digit hexadecimal index.

resource name

Specifies the name of the resource that has detected the error. For software errors, this is the name of a software component or an executable program. For hardware errors, this is the name of a device or system component. It does not indicate that the component is faulty or needs replacement instead, it is used to determine the appropriate diagnostic modules to be used to analyze the error.

detail_data

Specifies an array from 0 to **ERR_REC_MAX** bytes of user-supplied data. This data may be displayed by the **errpt** command in hexadecimal, alphanumeric, or binary form, according to the data_encoding fields in the error log template for this error id field.

Specifies the length in bytes of the **err_rec** structure, which is equal to the size of the error id and resource name fields plus the length in bytes of the detail data field.

Length

Return Values

- 0 The entry was logged successfully.
- The entry was not logged.

Files

/dev/error Provides standard device driver interfaces required by the error

log component.

/usr/include/sys/errids.h Contains definitions for error IDs.

/usr/include/sys/err_rec.h Contains structures defined as arguments to the errsave kernel

service and the errlog subroutine.

/var/adm/ras/errlog Maintains the system error log.

Related Information

The errclear, errdead, errinstall, errlogger, errmsg, errpt, errstop, and errupdate commands.

The errlog_open, errlog_close, errlog_find_first, errlog_find_next, errlog_find_sequence, errlog_set_direction, and errlog_write subroutines.

The /dev/error special file.

The **errdemon** daemon.

The errsave kernel service.

Error Logging Overview in Messages Guide and Reference.

errlog close Subroutine

Purpose

Closes an open error log file.

Syntax

library liberrlog.a

#include <sys/errlog.h>

int errlog close(handle) errlog_handle_t handle;

Description

The error log specified by the handle argument is closed. The handle must have been returned from a previous errlog_open call.

Return Values

Upon successful completion, the errlog close subroutine returns 0.

If an error occurs, the **errlog_close** subroutine returns **LE_ERR_INVARG**.

Related Information

The errlog_open, errlog_find_first, errlog_find_next, errlog_find_sequence, errlog_set_direction, errlog_write, and errlog subroutines.

errlog find first, errlog find next, and errlog find sequence **Subroutines**

Purpose

Retrieves an error log entry using supplied criteria.

Syntax

```
library liberrlog.a
#include <sys/errlog.h>
int errlog find first(handle, filter, result)
errlog handle t handle;
errlog match t *filter;
errlog_entry_t *result;
int errlog find next(handle, result)
errlog handle t handle;
errlog entry t *result;
int errlog find sequence(handle, sequence, result)
errlog handle t handle;
int sequence;
errlog entry t *result;
```

Description

The errlog_find_first subroutine finds the first occurrence of the search argument specified by filter using the direction specified by the errlog_set_direction subroutine. The reverse direction is used if none was specified. In other words, by default, entries are searched starting with the most recent entry.

The **errlog match t** structure, pointed to by the filter parameter, defines a test expression or set of expressions to be applied to each errlog entry.

If the value passed in the filter parameter is null, the errlog find first subroutine returns the first entry in the log, and the errlog find next subroutine can then be used to return subsequent entries. To read all log entries in the desired direction, open the log, then issue errlog find next calls.

To define a basic expression, **em field** must be set to the field in the errlog entry to be tested, **em op** must be set to the relational operator to be applied to that field, and either em_intvalue or em_strvalue must be set to the value to test against. Basic expressions may be combined by attaching them to em_left and **em right** of another **errlog match t** structure and setting **em op** of that structure to a binary or unary operator. These complex expressions may then be combined with other basic or complex expressions in the same fashion to build a tree that can define a filter of arbitrary complexity.

The errlog_find_next subroutine finds the next error log entry matching the criteria specified by a previous errlog_find_first call. The search continues in the direction specified by the errlog set direction subroutine or the reverse direction by default.

The **errlog find sequence** subroutine returns the entry matching the specified error log sequence number, found in the **el_sequence** field of the **errlog_entry** structure.

Parameters

The handle contains the handle returned by a prior call to errlog_open.

The filter parameter points to an **errlog match t** element defining the search argument, or the first of an argument tree.

The sequence parameter contains the sequence number of the entry to be retrieved.

The result parameter must point to the area to contain the returned error log entry.

Return Values

Upon successful completion, the errlog_find_first, errlog_find_next, and errlog_find_sequence subroutines return 0, and the memory referenced by result contains the found entry.

The following errors may be returned:

LE_ERR_INVARG LE_ERR_NOMEM LE ERR IO LE ERR DONE

A parameter error was detected. Memory could not be allocated. An i/o error occurred. No more entries were found.

Examples

The code below demonstrates how to search for all errlog entries in a date range and with a class of H (hardware) or **S** (software).

```
extern int
                  begintime, endtime;
errlog match t
                  beginstamp, endstamp, andstamp;
errlog_match_t
                hardclass, softclass, orclass;
errlog_match_t
                 andtop;
                  ret;
int
errlog entry t
                  result;
 * Select begin and end times
beginstamp.em op = LE OP GT;
                                            /* Expression 'A' */
beginstamp.em_field = LE_MATCH TIMESTAMP;
beginstamp.em intvalue=begintime;
endstamp.em op = LE OP LT;
                                            /* Expression 'B' */
endstamp.em field = LE MATCH TIMESTAMP;
endstamp.em intvalue=endtime;
                                            /* 'A' and 'B' */
andstamp.em op = LE OP AND;
andstamp.em left = &beginstamp;
andstamp.em right = &endstamp;
 * Select the classes we're interested in.
hardclass.em_op = LE_OP_EQ;
                                           /* Expression 'C' */
hardclass.em_field = LE_MATCH_CLASS;
hardclass.em_strvalue = "H";
softclass.em_op = LE_OP_EQ;
                                           /* Expression 'D' */
softclass.em field = LE MATCH CLASS;
softclass.em strvalue = "S";
                                           /* 'C' or 'D' */
orclass.em op = LE OP OR;
```

The errlog_find_first function will return the first entry matching filter. Successive calls to the errlog_find_next function will return successive entries that match the filter specified in the most recent call to the errlog_find_first function. When no more matching entries are found, the errlog_find_first and errlog_find_next functions will return the value LE_ERR_DONE.

Related Information

The errlog_open, errlog_close, errlog_set_direction, errlog_write, and errlog subroutines.

errlog_open Subroutine

Purpose

Opens an error log and returns a handle for use with other **liberrlog.a** functions.

Syntax

```
library liberrlog.a
#include <fcntl.h>
#include <sys/errlog.h>
int errlog_open(path, mode, magic, handle)
char *path;
int mode;
unsigned int magic;
errlog handle t *handle;
```

Description

The error log specified by the path argument will be opened using mode. The handle pointed to by the handle parameter must be used with subsequent operations.

Parameters

The path parameter specifies the path to the log file to be opened. If path is NULL, the default errlog file will be opened. The valid values for mode are the same as they are for the open system subroutine. They can be found in the **fcntl.h** files.

The **magic** argument takes the **LE_MAGIC** value, indicating which version of the **errlog_entry_t** structure this application was compiled with.

Return Values

Upon successful completion, the **errlog_open** subroutine returns a 0 and sets the memory pointed to by handle to a handle used by subsequent **liberrlog** operations.

Upon error, the **errlog_open** subroutine returns one of the following:

LE_ERR_INVARGA parameter error was detected.LE_ERR_NOFILEThe log file does not exist.

LE_ERR_NOMEM LE_ERR_IO LE ERR INVFILE Memory could not be allocated. An i/o error occurred. The file is not a valid error log.

Related Information

The errlog_close, errlog_find_first, errlog_find_next, errlog_find_sequence, errlog_set_direction, errlog_write, and errlog subroutines.

The /usr/include/fcntl.h include files found in AIX 5L Version 5.2 Files Reference.

errlog set direction Subroutine

Purpose

Sets the direction for the error log find functions.

Syntax

library liberrlog.a

#include <sys/errlog.h>

int errlog set direction(handle, direction) errlog handle t handle; int direction;

Description

The errlog_find_next and errlog_find_sequence subroutines search the error log starting with the most recent log entry and going backward in time, by default. The errlog set direction subroutine is used to alter this direction.

Parameters

The handle parameter must contain a handle returned by a previous errlog_open call.

The direction parameter must be LE_FORWARD or LE_REVERSE. LE_REVERSE is the default if the errlog_set_direction subroutine is not used.

Return Values

Upon successful completion, the errlog_set_direction subroutine returns 0.

If a parameter is invalid, the errlog_set_direction subroutine returns LE_ERR_INVARG.

Related Information

The errlog_open, errlog_close, errlog_find_first, errlog_find_next, errlog_find_sequence, errlog_write, and errlog subroutines.

errlog_write Subroutine

Purpose

Changes the previously read error log entry.

Syntax

```
library liberrlog.a
#include <sys/errlog.h>
int errlog_write(handle, entry)
errlog_handle_t handle;
errlog_entry_t *entry;
```

Description

The **errlog_write** subroutine is used to update the most recently read log entry. Neither the length nor the sequence number of the entry may be changed. The entry is simply updated in place.

If the **errlog_write** subroutine is used in a multi-threaded application, the program should obtain a lock around the read/write pair to avoid conflict.

Parameters

The handle parameter must contain a handle returned by a previous errlog_open call.

The entry parameter must point to an entry returned by the previous error log find function.

Return Values

Upon successful completion, the errlog_write subroutine returns 0.

If a parameter is invalid, the **errlog_write** subroutine returns **LE_ERR_INVARG**.

The **errlog_write** subroutine may also return one of the following:

LE_ERR_INVFILEThe data on file is invalid.LE_ERR_IOAn i/o error occurred.

LE_ERR_NOWRITEThe entry to be written didn't match the entry being

updated.

Related Information

The errlog_open, errlog_close, errlog_find_first, errlog_find_next, errlog_find_sequence, errlog_set_direction, and errlog subroutines.

The /usr/include/sys/errlog.h include file.

exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine

Purpose

Executes a file.

Library

Standard C Library (libc.a)

Syntax

#include <unistd.h>
extern
char **environ;

```
int execl (
Path,
Argument0 [, Argument1, ...], 0)
const char *Path, *Argument0, *Argument
1, ...;
int execle (
Path,
Argument0 [, Argument1, ...], 0,
EnvironmentPointer)
const
char *Path, *Argument0, *Argum
ent
1, ...;
char *const EnvironmentPointer[];
int execlp (
File,
Argument0 [, Argument1
, ...], 0)
const char *File, *Argument0, *Argument
1, ...;
int execv (
Path,
ArgumentV)
const char *Path;
char *const ArgumentV[];
int execve (
Path,
ArgumentV,
EnvironmentPointer)
const char *Path;
*const ArgumentV[], *EnvironmentPointer
[];
int execvp (
File,
ArgumentV)
const char *File;
char *const ArgumentV[];
int exect (
Path,
ArgumentV,
EnvironmentPointer)
char *Path, *ArgumentV, *EnvironmentPointer [];
```

Description

The **exec** subroutine, in all its forms, executes a new program in the calling process. The **exec** subroutine does not create a new process, but overlays the current program with a new one, which is called the new-process image. The new-process image file can be one of three file types:

- · An executable binary file in XCOFF file format. .
- An executable text file that contains a shell procedure (only the execlp and execvp subroutines allow this type of new-process image file).
- · A file that names an executable binary file or shell procedure to be run.

The new-process image inherits the following attributes from the calling process image: session membership, supplementary group IDs, process signal mask, and pending signals.

The last of the types mentioned is recognized by a header with the following syntax: #! Path [String]

The #! is the file magic number, which identifies the file type. The path name of the file to be executed is specified by the Path parameter. The String parameter is an optional character string that contains no tab or space characters. If specified, this string is passed to the new process as an argument in front of the name of the new-process image file. The header must be terminated with a new-line character. When called, the new process passes the Path parameter as Argument V[0]. If a String parameter is specified in the new process image file, the exec subroutine sets ArgumentV[0] to the String and Path parameter values concatenated together. The rest of the arguments passed are the same as those passed to the **exec** subroutine.

The exec subroutine attempts to cancel outstanding asynchronous I/O requests by this process. If the asynchronous I/O requests cannot be canceled, the application is blocked until the requests have completed.

The exec subroutine is similar to the load ("load Subroutine" on page 618) subroutine, except that the exec subroutine does not have an explicit library path parameter. Instead, the exec subroutine uses the LIBPATH environment variable. The LIBPATH variable is ignored when the program that the exec subroutine is run on has more privilege than the calling program, for example, the **suid** program.

The **exect** subroutine is included for compatibility with older programs being traced with the **ptrace** command. The program being executed is forced into hardware single-step mode.

Note: exect is not supported in 64-bit mode.

Note: Currently, a Graphics Library program cannot be overlaid with another Graphics Library program. The overlaying program can be a nongraphics program. For additional information, see the /usr/lpp/GL/README file.

Parameters

Path

Specifies a pointer to the path name of the new-process image file. If Network File System (NFS) is installed on your system, this path can cross into another node. Data is copied into local virtual memory before proceeding.

File

Argument0 [, Argument1, ...]

ArgumentV

EnvironmentPointer

Specifies a pointer to the name of the new-process image file. Unless the File parameter is a full path name, the path prefix for the file is obtained by searching the directories named in the PATH environment variable. The initial environment is supplied by the shell.

Note: The execlp subroutine and the execvp subroutine take File parameters, but the rest of the exec subroutines take Path parameters. (For information about the environment, see the **environment** miscellaneous facility and the **sh** command.) Point to null-terminated character strings. The strings constitute the argument list available to the new process. By convention, at least the Argument0 parameter must be present, and it must point to a string that is the same as the Path parameter or its last component.

Specifies an array of pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, the *ArgumentV* parameter must have at least one element, and it must point to a string that is the same as the *Path* parameter or its last component. The last element of the *ArgumentV* parameter is a null pointer. An array of pointers to null-terminated character strings. These strings constitute the environment for the new process. The last element of the EnvironmentPointer parameter is a null pointer.

When a C program is run, it receives the following parameters:

```
main (ArgumentCount, ArgumentV, EnvironmentPointer)
int ArgumentCount;
char *ArgumentV[], *EnvironmentPointer[
```

In this example, the ArgumentCount parameter is the argument count, and the ArgumentV parameter is an array of character pointers to the arguments themselves. By convention, the value of the ArgumentCount parameter is at least 1, and the Argument ([0] parameter points to a string containing the name of the new-process image file.

The main routine of a C language program automatically begins with a runtime start-off routine. This routine sets the **environ** global variable so that it points to the environment array passed to the program in EnvironmentPointer. You can access this global variable by including the following declaration in your program:

extern char **environ;

The exect, exect, and execvp subroutines use the environ global variable to pass the calling process current environment to the new process.

File descriptors open in the calling process remain open, except for those whose close-on-exec flag is set. For those file descriptors that remain open, the file pointer is unchanged. (For information about file control, see the fcntl.h file.)

The state-of-conversion descriptors and message-catalog descriptors in the new process image are undefined. For the new process, an equivalent of the setlocale subroutine, specifying the LC_ALL value for its category and the "C" value for its locale, is run at startup.

If the new program requires shared libraries, the exec subroutine finds, opens, and loads each of them into the new-process address space. The referenced counts for shared libraries in use by the issuer of the exec are decremented. Shared libraries are searched for in the directories listed in the LIBPATH environment variable. If any of these files is remote, the data is copied into local virtual memory.

The **exec** subroutines reset all caught signals to the default action. Signals that cause the default action continue to do so after the exec subroutines. Ignored signals remain ignored, the signal mask remains the same, and the signal stack state is reset. (For information about signals, see the sigaction subroutine.)

If the SetUserID mode bit of the new-process image file is set, the exec subroutine sets the effective user ID of the new process to the owner ID of the new-process image file. Similarly, if the SetGroupID mode bit of the new-process image file is set, the effective group ID of the new process is set to the group ID of the new-process image file. The real user ID and real group ID of the new process remain the same as those of the calling process. (For information about the SetID modes, see the chmod subroutine.)

At the end of the exec operation the saved user ID and saved group ID of the process are always set to the effective user ID and effective group ID, respectively, of the process.

When one or both of the set ID mode bits is set and the file to be executed is a remote file, the file user and group IDs go through outbound translation at the server. Then they are transmitted to the client node where they are translated according to the inbound translation table. These translated IDs become the user and group IDs of the new process.

Note: setuid and setgid bids on shell scripts do not affect user or group IDs of the process finally executed.

Profiling is disabled for the new process.

The new process inherits the following attributes from the calling process:

- Nice value (see the getpriority subroutine, setpriority subroutine, nice subroutine)
- · Process ID
- · Parent process ID
- Process group ID
- **semadj** values (see the **semop** subroutine)
- tty group ID (see the exit, atexit, or _exit subroutine, sigaction subroutine)
- trace flag (see request 0 of the ptrace subroutine)
- Time left until an alarm clock signal (see the incinterval subroutine, setitimer subroutine, and alarm subroutine)
- Current directory
- Root directory
- File-mode creation mask (see the umask subroutine)
- File size limit (see the **ulimit** subroutine)
- Resource limits (see the **getrlimit** subroutine, **setrlimit** subroutine, and **vlimit** subroutine)
- tms_utime, tms_stime, tms_cutime, and tms_ctime fields of the tms structure (see the times subroutine)
- Login user ID

Upon successful completion, the exec subroutines mark for update the st atime field of the file.

Examples

1. To run a command and pass it a parameter, enter:

```
execlp("ls", "ls", "-al", 0);
```

The **execlp** subroutine searches each of the directories listed in the **PATH** environment variable for the Is command, and then it overlays the current process image with this command. The execlp subroutine is not returned, unless the Is command cannot be executed.

Note: This example does not run the shell command processor, so operations interpreted by the shell, such as using wildcard characters in file names, are not valid.

2. To run the shell to interpret a command, enter:

```
execl("/usr/bin/sh", "sh", "-c", "ls -l *.c",
```

This runs the **sh** command with the **-c** flag, which indicates that the following parameter is the command to be interpreted. This example uses the execl subroutine instead of the execlp subroutine because the full path name /usr/bin/sh is specified, making a path search unnecessary.

Running a shell command in a child process is generally more useful than simply using the exec subroutine, as shown in this example. The simplest way to do this is to use the **system** subroutine.

3. The following is an example of a new-process file that names a program to be run:

```
#! /usr/bin/awk -f
{ for (i = NF; i > 0; --i) print i \}
```

If this file is named reverse, entering the following command on the command line:

reverse chapter1 chapter2

This command runs the following command:

/usr/bin/awk -f reverse chapter1 chapter2

Note: The exec subroutines use only the first line of the new-process image file and ignore the rest of it. Also, the awk command interprets the text that follows a # (pound sign) as a comment.

Return Values

Upon successful completion, the exec subroutines do not return because the calling process image is overlaid by the new-process image. If the **exec** subroutines return to the calling process, the value of -1 is returned and the errno global variable is set to identify the error.

Error Codes

If the exec subroutine is unsuccessful, it returns one or more of the following error codes:

EACCES The new-process image file is not an ordinary file.

EACCES The mode of the new-process image file denies execution permission.

ENOEXEC The **exec** subroutine is neither an **execlp** subroutine nor an **execvp** subroutine. The

new-process image file has the appropriate access permission, but the magic number in its

header is not valid.

ENOEXEC The new-process image file has a valid magic number in its header, but the header is

damaged or is incorrect for the machine on which the file is to be run.

ETXTBSY The new-process image file is a pure procedure (shared text) file that is currently open for

writing by some process.

The new process requires more memory than is allowed by the system-imposed maximum, **ENOMEM**

the MAXMEM compiler option.

E2BIG The number of bytes in the new-process argument list is greater than the system-imposed

limit. This limit is a system configurable value that can be set by superusers or system group

users using SMIT. Refer to Kernel Tunable Parameters for details.

The Path. ArgumentV, or EnvironmentPointer parameter points outside of the process address **EFAULT**

EPERM The SetUserID or SetGroupID mode bit is set on the process image file. The translation

tables at the server or client do not allow translation of this user or group ID.

If the **exec** subroutine is unsuccessful because of a condition requiring path name resolution, it returns one or more of the following error codes:

EACCES Search permission is denied on a component of the path prefix. Access could be denied due

to a secure mount.

EFAULT The Path parameter points outside of the allocated address space of the process.

EIO An input/output (I/O) error occurred during the operation.

ELOOP Too many symbolic links were encountered in translating the *Path* parameter.

ENAMETOOLONG A component of a path name exceeded 255 characters and the process has the disallow

truncation attribute (see the ulimit subroutine), or an entire path name exceeded 1023

characters.

ENOENT A component of the path prefix does not exist.

A symbolic link was named, but the file to which it refers does not exist. **ENOENT**

ENOENT The path name is null.

ENOTDIR A component of the path prefix is not a directory.

The root or current directory of the process is located in a virtual file system that has been **ESTALE**

unmounted.

In addition, some errors can occur when using the new-process file after the old process image has been overwritten. These errors include problems in setting up new data and stack registers, problems in mapping a shared library, or problems in reading the new-process file. Because returning to the calling process is not possible, the system sends the SIGKILL signal to the process when one of these errors occurs.

If an error occurred while mapping a shared library, an error message describing the reason for error is written to standard error before the signal SIGKILL is sent to the process. If a shared library cannot be mapped, the subroutine returns one of the following error codes:

ENOENT One or more components of the path name of the shared library file do not exist.

ENOTDIR A component of the path prefix of the shared library file is not a directory.

A component of a path name prefix of a shared library file exceeded 255 characters, or **ENAMETOOLONG**

an entire path name exceeded 1023 characters.

EACCES Search permission is denied for a directory listed in the path prefix of the shared library

EACCES The shared library file mode denies execution permission.

ENOEXEC The shared library file has the appropriate access permission, but a magic number in its

header is not valid.

ETXTBSY The shared library file is currently open for writing by some other process.

ENOMEM The shared library requires more memory than is allowed by the system-imposed

maximum.

ESTALE The process root or current directory is located in a virtual file system that has been

unmounted.

EPROCLIM If WLM is running, the limit on the number of processes, threads, or logins in the class

may have been met.

If NFS is installed on the system, the exec subroutine can also fail if the following is true:

ETIMEDOUT The connection timed out.

Related Information

The alarm ("getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329) or incinterval ("getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329) subroutine, chmod ("chmod or fchmod Subroutine" on page 124) or fchmod ("chmod or fchmod Subroutine" on page 124) subroutine, exit ("exit, atexit, _exit, or _Exit Subroutine" on page 203) subroutine, fcntl ("fcntl, dup, or dup2 Subroutine" on page 214) subroutine, fork ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, getrusage ("getrusage, getrusage64, times, or vtimes Subroutine" on page 360) or times ("getrusage, getrusage64, times, or

vtimes Subroutine" on page 360) subroutine, nice ("getpriority, setpriority, or nice Subroutine" on page 349) subroutine, profil ("profil Subroutine" on page 876) subroutine, ptrace ("ptrace, ptracex, ptrace64 Subroutine" on page 978) subroutine.

The **semop** subroutine, **settimer** ("gettimer, settimer, restimer, stime, or time Subroutine" on page 375) subroutine, sigaction, signal, or sigvec subroutine, shmat subroutine, system subroutine, ulimit subroutine, umask subroutine.

The **awk** command, **ksh** command, **sh** command.

The **environment** file.

The XCOFF object (a.out) file format.

The varargs macros.

Asynchronous I/O Overview in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

exit, atexit, _exit, or _Exit Subroutine

Purpose

Terminates a process.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
void exit ( Status)
int Status;
void _exit ( Status)
int Status;
void _Exit (Status)
int Status;
#include <sys/limits.h>
int atexit ( Function)
void (*Function) (void);
```

Description

The exit subroutine terminates the calling process after calling the standard I/O library _cleanup function to flush any buffered output. Also, it calls any functions registered previously for the process by the atexit subroutine. The atexit subroutine registers functions called at normal process termination for cleanup processing. Normal termination occurs as a result of either a call to the exit subroutine or a return statement in the main function.

Each function a call to the atexit subroutine registers must return. This action ensures that all registered functions are called.

Finally, the exit subroutine calls the exit subroutine, which completes process termination and does not return. The exit subroutine terminates the calling process and causes the following to occur:

The _Exit subroutine is functionally equivalent to the _exit subroutine. The _Exit subroutine does not call functions registered with atexit or any registered signal handlers. The way the subroutine is implemented determines whether open streams are flushed or closed, and whether temporary files are removed. The calling process is terminated with the consequences described below.

- All of the file descriptors, directory streams, conversion descriptors, and message catalog descriptors open in the calling process are closed.
- · If the parent process of the calling process is executing a wait or waitpid, and has not set its SA NOCLDWAIT flag nor set SIGCHLD to SIG IGN, it is notified of the calling process' termination and the low order eight bits (that is, bits 0377) of status are made available to it. If the parent is not waiting, the child's status is made available to it when the parent subsequently executes wait or waitpid.
- · If the parent process of the calling process is not executing a wait or waitpid, and has neither set its SA NOCLDWAIT flag nor set SIGCHLD to SIG IGN, the calling process is transformed into a zombie process. A zombie process is an inactive process that is deleted at some later time when its parent process executes wait or waitpid.
- Termination of a process does not directly terminate its children. The sending of a SIGHUP signal indirectly terminates children in some circumstances. This can be accomplished in one of two ways. If the implementation supports the SIGCHLD signal, a SIGCHLD is sent to the parent process. If the parent process has set its SA NOCLDWAIT flag, or set SIGCHLD to SIG IGN, the status is discarded, and the lifetime of the calling process ends immediately. If SA NOCLDWAIT is set, it is implementation defined whether a SIGCHLD signal is sent to the parent process.
- The parent process ID of all of the calling process' existing child processes and zombie processes are set to the process ID of an implementation defined system process.
- Each attached shared memory segment is detached and the value of shm nattch (see shmget) in the data structure associated with its shared memory ID is decremented by 1.
- For each semaphore for which the calling process has set a semadi value (see semop), that value is added to the semval of the specified semaphore.
- · If the process is a controlling process, the SIGHUP signal is sent to each process in the foreground process group of the controlling terminal belonging to the calling process.
- If the process is a controlling process, the controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process.
- · If the exit of the process causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, a SIGHUP signal followed by a SIGCONT signal is sent to each process in the newly orphaned process group.
- · All open named semaphores in the calling process are closed as if by appropriate calls to sem_close.
- Memory mappings that were created in the process are unmapped before the process is destroyed.
- Any blocks of typed memory that were mapped in the calling process are unmapped, as if the munmap subroutine was implicitly called to unmap them.
- All open message queue descriptors in the calling process are closed.
- Any outstanding cancelable asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled complete as if the Exit subroutine had not yet occurred, but any associated signal notifications are suppressed.
 - The **Exit** subroutine may block awaiting such I/O completion. The implementation defines whether any I/O is canceled, and which I/O may be canceled upon **Exit**.
- Threads terminated by a call to **Exit** do not invoke their cancelation cleanup handlers or per thread data destructors.
- · If the calling process is a trace controller process, any trace streams that were created by the calling process are shut down.

Parameters

Function

Status Indicates the status of the process. May be set to 0, EXIT_SUCCESS, EXIT_FAILURE, or any

other value, though only the least significant 8 bits are available to a waiting parent process. Specifies a function to be called at normal process termination for cleanup processing. You may

specify a number of functions to the limit set by the ATEXIT_MAX function, which is defined in

the sys/limits.h file. A pushdown stack of functions is kept so that the last function registered is

the first function called.

Return Values

Upon successful completion, the atexit subroutine returns a value of 0. Otherwise, a nonzero value is returned. The exit and exit subroutines do not return a value.

Related Information

"acct Subroutine" on page 8, "lockfx, lockf, flock, or lockf64 Subroutine" on page 628, "lockfx, lockf, flock, or lockf64 Subroutine" on page 628, "lockfx, lockf, flock, or lockf64 Subroutine" on page 628, and "getrusage, getrusage64, times, or vtimes Subroutine" on page 360.

longimp Subroutine, semop Subroutine, shmget Subroutine, sigaction, sigvec, or signal Subroutine, and wait, waitpid, or wait3 Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Asynchronous I/O Overview in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

unistd.h in AIX 5L Version 5.2 Files Reference.

exp, expf, or expl Subroutine

Purpose

Computes exponential, logarithm, and power functions.

Libraries

IEEE Math Library (libm.a) or System V Math Library (libmsaa.a)

Syntax

#include <math.h> double exp (x)double x; float expf (x)float x; long double expl (x)long double x;

Description

These subroutines are used to compute exponential, logarithm, and power functions.

The **exp**, **expf**, and **expl** subroutines returns exp(x).

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept(FE_ALL_EXCEPT)** before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Parameters

- x Specifies some double-precision floating-point value.
- y Specifies some double-precision floating-point value.

Return Values

Upon successful completion, the exp, expf, and expl subroutines return the exponential value of x.

If the correct value would cause overflow, a range error occurs and the **exp**, **expf**, and **expl** subroutine returns the value of the macro **HUGE VAL**, **HUGE VALF** and **HUGE VALL**, respectively.

If the correct value would cause underflow, and is not representable, a range error may occur, and either 0.0 (if supported), or an implementation-defined value is returned.

If x is NaN, a NaN is returned.

If x is ± 0 , 1 is returned.

If x is -Inf, +0 is returned.

If x is +Inf, x is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

Error Codes

When using the libm.a library:

exp If the correct value would overflow, the exp subroutine returns a HUGE_VAL value and the errno global variable is set to a ERANGE value.

When using libmsaa.a(-lmsaa):

exp If the correct value would overflow, the exp subroutine returns a HUGE_VAL value. If the correct

value would underflow, the exp subroutine returns 0. In both cases errno is set to ERANGE.

expl If the correct value would overflow, the expl subroutine returns a HUGE_VAL value. If the correct

value would underflow, the expl subroutine returns 0. In both cases errno is set to ERANGE.

expl If the correct value overflows, the expl subroutine returns a HUGE VAL value and errno is set to

ERANGE.

These error-handling procedures may be changed with the **matherr** subroutine when using the **libmsaa.a** library.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, _class, finite, isnan, or unordered Subroutines" on page 138.

The matherr ("matherr Subroutine" on page 665) subroutine, sinh, cosh, or tanh subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

128-Bit long double Floating-Point Format in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

math.h in AIX 5L Version 5.2 Files Reference.

exp2, exp2f, or exp2l Subroutine

Purpose

Computes the base 2 exponential.

Syntax

```
#include <math.h>
double exp2(x)
double x;
float exp2f (x)
float x;
long double exp21 (x)
long double x;
```

Description

The exp2, exp2f, and exp2l subroutines compute the base 2 exponential of the x parameter.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept (FE ALL EXCEPT) before calling these subroutines. On return, if errno is nonzero or fetestexcept (FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the base 2 exponential to be computed.

Return Values

Upon successful completion, the exp2, exp2f, or exp2l subroutine returns 2x.

If the correct value causes overflow, a range error occurs and the exp2, exp2f, and exp2l subroutines return the value of the macro (HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively).

If the correct value causes underflow and is not representable, a range error occurs, and 0.0 is returned.

If x is NaN, NaN is returned.

If x is ± 0 , 1 is returned.

If x is -Inf, 0 is returned.

If x is +Inf, x is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

Related Information

math.h in AIX 5L Version 5.2 Files Reference.

expm1, expm1f, or expm1l Subroutine

Purpose

Computes exponential functions.

Syntax

```
#include <math.h>
float expm1f (x)
float x;
long double expm1l (x)
long double x;
double expm1 (x)
double x;
```

Description

The **expm1f**, **expm1l**, and **expm1** subroutines compute e^{x} - 1.0.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these functions. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID** | **FE_DIVBYZERO** | **FE_OVERFLOW** | **FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

x Specifies the value to be computed.

Return Values

Upon successful completion, the **expm1f**, **expm1f**, and **expm1** subroutines return e^{x} 1.0.

If the correct value would cause overflow, a range error occurs and the **expm1f**, **expm1l**, and **expm1** subroutines return the value of the macro **HUGE_VALF**, **HUGE_VALL**, and **HUGE_VAL**, respectively.

If x is NaN, a NaN is returned.

If x is ± 0 , ± 0 is returned.

If x is -Inf, -1 is returned.

If x is +Inf, x is returned.

If x is subnormal, a range error may occur and x is returned.

Related Information

"exp, expf, or expl Subroutine" on page 205, "feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, "ilogbf, ilogbl, or ilogb Subroutine" on page 432, and "log, logf, or logl Subroutine" on page 635.

math.h in AIX 5L Version 5.2 Files Reference.

fabsf, fabsl, or fabs Subroutine

Purpose

Determines the absolute value.

Syntax

```
#include <math.h>
float fabsf (x)
float x;
long double fabsl (x)
long double x;
double fabs (x)
double x;
```

Description

The fabsf, fabsl, and fabs subroutines compute the absolute value of the x parameter, |x|.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the fabsf, fabsl, and fabs subroutines return the absolute value of x.

If x is NaN, a NaN is returned.

If x is ± 0 , ± 0 is returned.

If x is $\pm \ln f$, $+\ln f$ is returned.

Related Information

The "class, _class, finite, isnan, or unordered Subroutines" on page 138.

math.h in AIX 5L Version 5.2 Files Reference.

fattach Subroutine

Purpose

Attaches a STREAMS-based file descriptor to a file.

Library

Standard C Library (libc.a)

Syntax

#include <stropts.h>
int fattach(int fildes, const char *path);

Description

The **fattach** subroutine attaches a STREAMS-based file descriptor to a file, effectively associating a pathname with *fildes*. The *fildes* argument must be a valid open file descriptor associated with a STREAMS file. The *path* argument points to a pathname of an existing file. The process must have appropriate privileges, or must be the owner of the file named by *path* and have write permission. A successful call to **fattach** subroutine causes all pathnames that name the file named by *path* to name the STREAMS file associated with *fildes*, until the STEAMS file is detached from the file. A STREAMS file can be attached to more than one file and can have several pathnames associated with it.

The attributes of the named STREAMS file are initialized as follows: the permissions, user ID, group ID, and times are set to those of the file named by *path*, the number of links is set to 1, and the size and device identifier are set to those of the STREAMS file associated with *fildes*. If any attributes of the named STREAMS file are subsequently changed (for example, by **chmod** subroutine), neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildes* refers are affected.

File descriptors referring to the underlying file, opened prior to an **fattach** subroutine, continue to refer to the underlying file.

Parameters

fildes A file descriptor identifying an open STREAMS-based object. path An existing pathname which will be associated with fildes.

Return Value

Successful completion.

-1 Not successful and *errno* set to one of the following.

Errno Value

EACCES Search permission is denied for a component of the path prefix, or the process is the owner

of path but does not have write permission on the file named by path.

EBADF The file referred to by *fildes* is not an open file descriptor.

ENOENT A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR A component of the path prefix is not a directory.

EPERM The effective user ID of the process is not the owner of the file named by *path* and the

process does not have appropriate privilege.

EBUSY The file named by *path* is currently a mount point or has a STREAMS file attached to it.

ENAMETOOLONG The size of path exceeds {PATH_MAX}, or a component of path is longer than

{NAME_MAX}.

ELOOP Too many symbolic links wer encountered in resolving *path*. **EINVAL** The *fildes* argument does not refer to a STREAMS file.

ENOMEM Insufficient storage space is available.

Related Specifics

The **fdetach** ("fdetach Subroutine" on page 221) subroutine, **isastream** subroutine.

fchdir Subroutine

Purpose

Directory pointed to by the file descriptor becomes the current working directory.

Library

Standard C Library (libc.a)

Syntax

#include <unistd.h>

int fchdir (int Fildes)

Description

The fchdir subroutine causes the directory specified by the Fildes parameter to become the current working directory.

Parameter

Fildes A file descriptor identifying an open directory obtained from a call to the open subroutine.

Return Values

Successful completion

-1 Not successful and errno set to one of the following.

Error Codes

EACCES Search access if denied.

EBADF The file referred to by Fildes is not an open file descriptor. **ENOTDIR** The open file descriptor does not refer to a directory.

Related Information

The **chdir** ("chdir Subroutine" on page 123) subroutine, **chroot** ("chroot Subroutine" on page 131) subroutine, open ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine.

fclear or fclear64 Subroutine

Purpose

Makes a hole in a file.

Library

Standard C Library (libc.a)

Syntax

```
off t fclear ( FileDescriptor, NumberOfBytes)
int FileDescriptor;
off_t NumberOfBytes;
off64 t fclear64 ( FileDescriptor, NumberOfBytes)
int FileDescriptor;
off64_t NumberOfBytes;
```

Description

The fclear and fclear64 subroutines zero the number of bytes specified by the NumberOfBytes parameter starting at the current file pointer for the file specified in the FileDescriptor parameter. If Network File System (NFS) is installed on your system, this file can reside on another node.

The fclear subroutine can only clear up to OFF_MAX bytes of the file while fclear64 can clear up to the maximum file size.

The fclear and fclear64 subroutines cannot be applied to a file that a process has opened with the O DEFER mode.

Successful completion of the fclear and fclear64 subroutines clear the SetUserID bit (S ISUID) of the file if any of the following are true:

- The calling process does not have root user authority.
- The effective user ID of the calling process does not match the user ID of the file.
- The file is executable by the group (S IXGRP) or others (S IXOTH).

This subroutine also clears the SetGroupID bit (S_ISGID) if:

- The file does not match the effective group ID or one of the supplementary group IDs of the process,
- The file is executable by the owner (S_IXUSR) or others (S_IXOTH).

Note: Clearing of the SetUserID and SetGroupID bits can occur even if the subroutine fails because the data in the file was modified before the error was detected.

In the large file enabled programming environment, fclear is redefined to be fclear64.

Parameters

FileDescriptor Indicates the file specified by the FileDescriptor parameter must be open for writing. The

> FileDescriptor is a small positive integer used instead of the file name to identify a file. This function differs from the logically equivalent write operation in that it returns full

blocks of binary zeros to the file system, constructing holes in the file.

NumberOfBytes Indicates the number of bytes that the seek pointer is advanced. If you use the fclear

> and fclear64 subroutines past the end of a file, the rest of the file is cleared and the seek pointer is advanced by NumberOfBytes. The file size is updated to include this new hole, which leaves the current file position at the byte immediately beyond the new end-of-file

pointer.

Return Values

Upon successful completion, a value of NumberOfBytes is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The fclear and fclear64 subroutines fail if one or more of the following are true:

FIO I/O error.

EBADF The FileDescriptor value is not a valid file descriptor open for writing.

The file is not a regular file. **EINVAL**

EMFILE The file is mapped **O_DEFER** by one or more processes.

The write operation in the fclear subroutine failed due to an enforced write lock on the file. **EAGAIN**

EFBIG The current offset plus NumberOfBytes is exceeds the offset maximum established in the open

file description associated with FileDescriptor.

EFBIG An attempt was made to write a file that exceeds the process' file size limit or the maximum file

> size. If the user has set the environment variable XPG_SUS_ENV=ON prior to execution of the process, then the SIGXFSZ signal is posted to the process when exceeding the process' file size

If NFS is installed on the system the fclear and fclear64 subroutines can also fail if the following is true:

ETIMEDOUT The connection timed out.

Related Information

The open, openx, or creat ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine, truncate or ftruncate subroutines.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fclose or fflush Subroutine

Purpose

Closes or flushes a stream.

Library

Standard C Library (libc.a)

Syntax

#include <stdio.h>

int fclose (Stream) FILE *Stream;

int fflush (Stream) FILE *Stream;

Description

The **fclose** subroutine writes buffered data to the stream specified by the *Stream* parameter, and then closes the stream. The fclose subroutine is automatically called for all open files when the exit subroutine is invoked.

The **fflush** subroutine writes any buffered data for the stream specified by the *Stream* parameter and leaves the stream open. The fflush subroutine marks the st ctime and st mtime fields of the underlying file for update.

If the Stream parameter is a null pointer, the fflush subroutine performs this flushing action on all streams for which the behavior is defined.

Parameters

Stream Specifies the output stream.

Return Values

Upon successful completion, the fclose and fflush subroutines return a value of 0. Otherwise, a value of EOF is returned.

Error Codes

If the fclose and fflush subroutines are unsuccessful, the following errors are returned through the errno global variable:

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor underlying the *Stream* parameter and the

process would be delayed in the write operation.

EBADF The file descriptor underlying Stream is not valid.

EFBIG An attempt was made to write a file that exceeds the process' file size limit or the maximum file

size. See the ulimit subroutine.

EFBIG The file is a regular file and an attempt was made to write at or beyond the offset maximum

associated with the corresponding stream.

EINTR The **fflush** subroutine was interrupted by a signal.

EIO The process is a member of a background process group attempting to write to its controlling

> terminal, the TOSTOP signal is set, the process is neither ignoring nor blocking the SIGTTOU signal and the process group of the process is orphaned. This error may also be returned under

implementation-dependent conditions.

ENOSPC No free space remained on the device containing the file.

An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **EPIPE**

SIGPIPE signal is sent to the process.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the

device

Related Information

The close ("close Subroutine" on page 141) subroutine, exit, atexit, or _exit ("exit, atexit, _exit, or _Exit Subroutine" on page 203) subroutine, fopen, freopen, or fdopen ("fopen, fopen64, freopen64 or fdopen Subroutine" on page 245) subroutine, setbuf, setbuff, setbuffer, or setlinebuf subroutine.

Input and Output Handling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fcntl, dup, or dup2 Subroutine

Purpose

Controls open file descriptors.

Library

Standard C Library (libc.a)

Syntax

#include <fcntl.h>

int fcntl (FileDescriptor, Command, Argument) int FileDescriptor, Command, Argument;

#include <unistd.h>

int dup2(Old, New) int Old, New;

int dup(FileDescriptor) int FileDescriptor;

Description

The **fcntl** subroutine performs controlling operations on the open file specified by the *FileDescriptor* parameter. If Network File System (NFS) is installed on your system, the open file can reside on another node. The **fcntl** subroutine is used to:

- · Duplicate open file descriptors.
- · Set and get the file-descriptor flags.
- · Set and get the file-status flags.
- Manage record locks.
- Manage asynchronous I/O ownership.
- · Close multiple files.

The **fcntl** subroutine can provide the same functions as the **dup** and **dup2** subroutines.

If FileDescriptor refers to a terminal device or socket, then asynchronous I/O facilities can be used. These facilities are normally enabled by using the ioctl subroutine with the FIOASYNC, FIOSETOWN, and FIOGETOWN commands. However, a BSD-compatible mechanism is also available if the application is linked with the libbsd.a library.

When using the libbsd.a library, asynchronous I/O is enabled by using the F_SETFL command with the FASYNC flag set in the Argument parameter. The F_GETOWN and F_SETOWN commands get the current asynchronous I/O owner and set the asynchronous I/O owner.

All applications containing the **fcntl** subroutine must be complied with **BSD** set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD libbsd.a library.

General Record Locking Information

A lock is either an enforced or advisory lock and either a read or a write lock.

Attention: Buffered I/O does not work properly when used with file locking. Do not use the standard I/O package routines on files that are going to be locked.

For a lock to be an enforced lock, the Enforced Locking attribute of the file must be set; for example, the S_ENFMT bit must be set, but the S_IXGRP, S_IXUSR, and S_IXOTH bits must be clear. Otherwise, the lock is an advisory lock. A given file can have advisory or enforced locks, but not both. The description of the **sys/mode.h** file includes a description of file attributes.

When a process holds an enforced lock on a section of a file, no other process can access that section of the file with the read or write subroutine. In addition, the open ("open, openx, open64, creat, or creat64 Subroutine" on page 763) and ftruncate subroutines cannot truncate the locked section of the file, and the fclear ("fclear or fclear64 Subroutine" on page 211) subroutine cannot modify the locked section of the file. If another process attempts to read or modify the locked section of the file, the process either sleeps until the section is unlocked or returns with an error indication.

When a process holds an advisory lock on a section of a file, no other process can lock that section of the file (or an overlapping section) with the **fcntl** subroutine. (No other subroutines are affected.) As a result, processes must voluntarily call the **fcntl** subroutine in order to make advisory locks effective.

When a process holds a read lock on a section of a file, other processes can also set read locks on that section or on subsets of it. Read locks are also called shared locks.

A read lock prevents any other process from setting a write lock on any part of the protected area. If the read lock is also an enforced lock, no other process can modify the protected area.

The file descriptor on which a read lock is being placed must have been opened with read access.

When a process holds a write lock on a section of a file, no other process can set a read lock or a write lock on that section. Write locks are also called exclusive locks. Only one write lock and no read locks can exist for a specific section of a file at any time.

If the lock is also an enforced lock, no other process can read or modify the protected area.

The following general rules about file locking apply:

- · Changing or unlocking part of a file in the middle of a locked section leaves two smaller sections locked at each end of the originally locked section.
- If the calling process holds a lock on a file, that lock can be replaced by later calls to the fcntl subroutine.
- · All locks associated with a file for a given process are removed when the process closes any file descriptor for that file.
- · Locks are not inherited by a child process after a fork ("fork, f_fork, or vfork Subroutine" on page 248) subroutine is run.

Note: Deadlocks due to file locks in a distributed system are not always detected. When such deadlocks can possibly occur, the programs requesting the locks should set time-out timers.

Locks can start and extend beyond the current end of a file but cannot be negative relative to the beginning of the file. A lock can be set to extend to the end of the file by setting the 1 1en field to 0. If such a lock also has the 1 start and 1 whence fields set to 0, the whole file is locked. The 1 len, 1 start, and 1 whence locking fields are part of the **flock** structure.

Note: The following description applies to AIX 4.3 and later releases.

When an application locks a region of a file using the 32 bit locking interface (F_SETLK), and the last byte of the lock range includes MAX OFF (2 Gb - 1), then the lock range for the unlock request will be extended to include MAX END (2 ^ ^ 63 - 1).

Parameters

Specifies an open file descriptor obtained from a successful call to the open, fcntl, or FileDescriptor

pipe ("pipe Subroutine" on page 815) subroutine. File descriptors are small positive

integers used (instead of file names) to identify files.

Specifies a variable whose value sets the function specified by the Command parameter. Argument

When dealing with file locks, the Argument parameter must be a pointer to the FLOCK

structure.

Command

Specifies the operation performed by the fcntl subroutine. The fcntl subroutine can duplicate open file descriptors, set file-descriptor flags, set file descriptor locks, set process IDs, and close open file descriptors.

Duplicating File Descriptors

F DUPFD Returns a new file descriptor as follows:

- · Lowest-numbered available file descriptor greater than or equal to the Argument parameter
- · Same object references as the original file
- · Same file pointer as the original file (that is, both file descriptors share one file pointer if the object is
- Same access mode (read, write, or read-write)
- Same file status flags (That is, both file descriptors share the same file status flags.)
- · The close-on-exec flag (FD_CLOEXEC bit) associated with the new file descriptor is cleared

Setting File-Descriptor Flags

F_GETFD

Gets the close-on-exec flag (FD CLOEXEC bit) that is associated with the file descriptor specified by the FileDescriptor parameter. The Argument parameter is ignored. File descriptor flags are associated with a single file descriptor, and do not affect others associated with the same file.

F SETFD

Assigns the value of the Argument parameter to the close-on-exec flag (FD_CLOEXEC bit) that is associated with the FileDescriptor parameter. If the FD CLOEXEC flag value is 0, the file remains open across any calls to exec subroutines; otherwise, the file will close upon the successful execution of an exec subroutine.

F_GETFL

Gets the file-status flags and file-access modes for the open file description associated with the file descriptor specified by the FileDescriptor parameter. The open file description is set at the time the file is opened and applies only to those file descriptors associated with that particular call to the file. This open file descriptor does not affect other file descriptors that refer to the same file with different open file descriptions.

The file-status flags have the following values:

O APPEND

Set append mode.

O NONBLOCK

No delay.

The file-access modes have the following values:

O RDONLY

Open for reading only.

O_RDWR

Open for reading and writing.

O_WRONLY

Open for writing only.

The file access flags can be extracted from the return value using the O_ACCMODE mask, which is defined in the fcntl.h file.

F_SETFL

Sets the file status flags from the corresponding bits specified by the *Argument* parameter. The file-status flags are set for the open file description associated with the file descriptor specified by the *FileDescriptor* parameter. The following flags may be set:

- O APPEND or FAPPEND
- O_NDELAY or FNDELAY
- O_NONBLOCK or FNONBLOCK
- O_SYNC or FSYNC
- FASYNC

The **O_NDELAY** and **O_NONBLOCK** flags affect only operations against file descriptors derived from the same **open** subroutine. In BSD, these operations apply to all file descriptors that refer to the object.

Setting File Locks

F GETLK

Gets information on the first lock that blocks the lock described in the **flock** structure. The *Argument* parameter should be a pointer to a type **struct flock**, as defined in the **flock.h** file. The information retrieved by the **fcntl** subroutine overwrites the information in the **struct flock** pointed to by the *Argument* parameter. If no lock is found that would prevent this lock from being created, the structure is left unchanged, except for lock type (1 type) which is set to **F_UNLCK**.

F_SETLK

Sets or clears a file-segment lock according to the lock description pointed to by the *Argument* parameter. The *Argument* parameter should be a pointer to a type **struct flock**, which is defined in the **flock.h** file. The **F_SETLK** option is used to establish read (or shared) locks (**F_RDLCK**), or write (or exclusive) locks (**F_WRLCK**), as well as to remove either type of lock (**F_UNLCK**). The lock types are defined by the **fcntl.h** file. If a shared or exclusive lock cannot be set, the **fcntl** subroutine returns immediately.

F_SETLKW

Performs the same function as the **F_SETLK** option unless a read or write lock is blocked by existing locks, in which case the process sleeps until the section of the file is free to be locked. If a signal that is to be caught is received while the **fcntl** subroutine is waiting for a region, the **fcntl** subroutine is interrupted, returns a -1, sets the **errno** global variable to **EINTR**. The lock operation is not done.

F_GETLK64

Gets information on the first lock that blocks the lock described in the **flock64** structure. The *Argument* parameter should be a pointer to an object of the type **struct flock64**, as defined in the **flock.h** file. The information retrieved by the **fcntl** subroutine overwrites the information in the **struct flock64** pointed to by the *Argument* parameter. If no lock is found that would prevent this lock from being created, the structure is left unchanged, except for lock type (1_type) which is set to **F_UNLCK**.

F SETLK64

Sets or clears a file-segment lock according to the lock description pointed to by the *Argument* parameter. The *Argument* parameter should be a pointer to a type **struct flock64**, which is defined in the **flock.h** file. The **F_SETLK** option is used to establish read (or shared) locks (**F_RDLCK**), or write (or exclusive) locks (**F_WRLCK**), as well as to remove either type of lock (**F_UNLCK**). The lock types are defined by the **fcntl.h** file. If a shared or exclusive lock cannot be set, the **fcntl** subroutine returns immediately.

F_SETLKW64

Performs the same function as the **F_SETLK** option unless a read or write lock is blocked by existing locks, in which case the process sleeps until the section of the file is free to be locked. If a signal that is to be caught is received while the **fcntl** subroutine is waiting for a region, the **fcntl** subroutine is interrupted, returns a -1, sets the **errno** global variable to **EINTR**. The lock operation is not done.

Setting Process ID

F_GETOWN

Gets the process ID or process group currently receiving **SIGIO** and **SIGURG** signals. Process groups are returned as negative values.

F_SETOWN

Sets the process or process group to receive **SIGIO** and **SIGURG** signals. Process groups are specified by supplying a negative *Argument* value. Otherwise, the *Argument* parameter is interpreted as a process ID.

Closing File Descriptors

F CLOSEM Closes all file descriptors from FileDescriptor up to the number specified by the OPEN MAX value.

Old Specifies an open file descriptor.

New Specifies an open file descriptor that is returned by the **dup2** subroutine.

Compatibility Interfaces

The lockfx Subroutine

The fcntl subroutine functions similar to the lockfx subroutine, when the Command parameter is F_SETLKW, or F_GETLK, and when used in the following way:

fcntl (FileDescriptor, Command, Argument)

is equivalent to:

lockfx (FileDescriptor, Command, Argument)

The dup and dup2 Subroutines

The **fcntl** subroutine functions similar to the **dup** and **dup2** subroutines, when used in the following way: dup (FileDescriptor)

```
is equivalent to:
```

```
fcntl (FileDescriptor, F_DUPFD, 0)
dup2 (01d, New)
```

is equivalent to:

```
close (New):
fcntl(Old, F DUPFD, New)
```

The dup and dup2 subroutines differ from the fcntl subroutine in the following ways:

- If the file descriptor specified by the New parameter is greater than or equal to OPEN MAX, the dup2 subroutine returns a -1 and sets the errno variable to EBADF.
- If the file descriptor specified by the Old parameter is valid and equal to the file descriptor specified by the New parameter, the dup2 subroutine will return the file descriptor specified by the New parameter, without closing it.
- If the file descriptor specified by the Old parameter is not valid, the dup2 subroutine will be unsuccessful and will not close the file descriptor specified by the New parameter.
- The value returned by the dup and dup2 subroutines is equal to the New parameter upon successful completion; otherwise, the return value is -1.

Return Values

Upon successful completion, the value returned depends on the value of the Command parameter, as follows:

Command	Return Value
F_DUPFD	A new file descriptor
F_GETFD	The value of the flag (only the FD_CLOEXEC bit is defined)
F_SETFD	A value other than -1
F_GETFL	The value of file flags
F_SETFL	A value other than -1
F_GETOWN	The value of descriptor owner
F_SETOWN	A value other than -1

F GETLK A value other than -1 F_SETLK A value other than -1 F SETLKW A value other than -1 F CLOSEM A value other than -1.

If the fcntl subroutine fails, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **fcntl** subroutine is unsuccessful if one or more of the following are true:

EACCES The Command argument is F SETLK; the type of lock is a shared or exclusive lock and the

> segment of a file to be locked is already exclusively-locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or

exclusive-locked by another process.

EBADF The *FileDescriptor* parameter is not a valid open file descriptor.

EDEADLK The Command argument is F_SETLKW; the lock is blocked by some lock from another process

and putting the calling process to sleep, waiting for that lock to become free would cause a

deadlock.

EMFILE The Command parameter is F_DUPFD, and the maximum number of file descriptors are currently

open (OPEN_MAX).

EINVAL The Command parameter is **F_DUPFD**, and the Argument parameter is negative or greater than or

equal to **OPEN_MAX**.

EINVAL An illegal value was provided for the Command parameter.

EINVAL An attempt was made to lock a fifo or pipe.

ESRCH The value of the Command parameter is F_SETOWN, and the process ID specified as the

Argument parameter is not in use.

EINTR The Command parameter was F_SETLKW and the process received a signal while waiting to

acquire the lock.

EOVERFLOW The Command parameter was F_GETLK and the block lock could not be represented in the flock

structure.

The **dup** and **dup2** subroutines fail if one or both of the following are true:

EBADF The Old parameter specifies an invalid open file descriptor or the New parameter specifies a file

descriptor that is out of range.

EMFILE The number of file descriptors exceeds the OPEN_MAX value or there is no file descriptor above the

value of the New parameter.

If NFS is installed on the system, the **fcntl** subroutine can fail if the following is true:

ETIMEDOUT The connection timed out.

Related Information

The close ("close Subroutine" on page 141) subroutine, execl, execve, execle, execve, execlp, execvp, or exect ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutines, fork ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, ioctl or ioctlx ("ioctl, ioctlx, ioctl32, or ioctl32x Subroutine" on page 459) subroutine, lockf ("lockfx, lockf, flock, or lockf64 Subroutine" on page 628) subroutine, open, openx, or creat ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutines, read subroutine, write subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fdetach Subroutine

Purpose

Detaches STREAMS-based file from the file to which it was attached.

Library

Standard C Library (libc.a)

Syntax

#include <stropts.h> int fdetach(const char *path);

Parameters

path Pathname of a file previous associated with a STREAMS-based object using the **fattach** subroutine.

Description

The fdetach subroutine detaches a STREAMS-based file from the file to which it was attached by a previous call to fattach subroutine. The path argument points to the pathname of the attached STREAMS file. The process must have appropriate privileges or be the owner of the file. A successful call to **fdetach** subroutine causes all pathnames that named the attached STREAMS file to again name the file to which the STREAMS file was attached. All subsequent operations on path will operate on the underlying file and not on the STREAMS file.

All open file descriptors established while the STREAMS file was attached to the file referenced by path will still refer to the STREAMS file after the fdetach subroutine has taken effect.

If there are no open file descriptors or other references to the STREAMS file, then a successful call to fdetach subroutine has the same effect as performing the last close subroutine on the attached file.

The **umount** command may be used to detach a file name if an I application exits before performing fdetach subroutine.

Return Value

0 Successful completion.

Not successful and errno set to one of the following.

Errno Value

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

EPERM The effective user ID is not the owner of path and the process does not have appropriate

privileges.

ENOTDIR A component of the path prefix is not a directory.

ENOENT A component of path parameter does not name an existing file or path is an empty string.

EINVAL The path parameter names a file that is not currently attached.

ENAMETOOLONG The size of path parameter exceeds {PATH_MAX}, or a component of path is longer than

{NAME_MAX}.

ELOOP Too many symbolic links were encountered in resolving the *path* parameter.

ENOMEM Insufficient storage space is available.

Related Information

The fattach ("fattach Subroutine" on page 209) subroutine, isastream subroutine.

fdim, fdimf, or fdiml Subroutine

Purpose

Computes the positive difference between two floating-point numbers.

Syntax

```
#include <math.h>
double fdim (x, y)
double x;
double y;
float fdimf (x, y)
float x;
float y;
long double fdiml (x, y)
long double x;
long double y;
```

Description

The **fdim**, **fdimf**, and **fdiml** subroutines determine the positive difference between their arguments. If x is greater than y, x - y is returned. If x is less than or equal to y, +0 is returned.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutyines. Upon return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

- Specifies the value to be computed.
- Specifies the value to be computed. У

Return Values

Upon successful completion, the fdim, fdimf, and fdiml subroutines return the positive difference value.

If x-y is positive and overflows, a range error occurs and the fdim, fdimf, and fdiml subroutines return the value of the macro HUGE VAL, HUGE VALF, and HUGE VALL, respectively.

If x-y is positive and underflows, a range error may occur, and 0.0 is returned.

If x or y is NaN, a NaN is returned.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, "fmax, fmaxf, or fmaxl Subroutine" on page 238, and "madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout, omout, fmout, m_out, sdiv, or itom Subroutine" on page 661.

math.h in AIX 5L Version 5.2 Files Reference.

feclearexcept Subroutine

Purpose

Clears floating-point exceptions.

Syntax

```
#include <fenv.h>
int feclearexcept (excepts)
int excepts;
```

Description

The feclearexcept subroutine attempts to clear the supported floating-point exceptions represented by the excepts parameter.

Parameters

excepts

Specifies the supported floating-point exception to be cleared.

Return Values

If the excepts parameter is zero or if all the specified exceptions were successfully cleared, the feclearexcept subroutine returns zero. Otherwise, it returns a nonzero value.

Related Information

"fegetexceptflag or fesetexceptflag Subroutine" on page 224, "feraiseexcept Subroutine" on page 228, and "fetestexcept Subroutine" on page 231

fegetenv or fesetenv Subroutine

Purpose

Gets and sets the current floating-point environment.

Syntax

```
#include <fenv.h>
int fegetenv (envp)
fenv_t *envp;
int fesetenv (envp)
const fenv_t *envp;
```

Description

The **fegetenv** subroutine stores the current floating-point environment in the object pointed to by the *envp* parameter.

The fesetenv subroutine attempts to establish the floating-point environment represented by the object pointed to by the envp parameter. The envp parameter points to an object set by a call to the fegetenv or feholdexcept subroutines, or equal a floating-point environment macro. The fesetenv subroutine does not raise floating-point exceptions. It only installs the state of the floating-point status flags represented through its argument.

Parameters

envp

Points to an object set by a call to the fegetenv or feholdexcept subroutines, or equal a floating-point environment macro.

Return Values

If the representation was successfully stored, the fegetenv subroutine returns zero. Otherwise, it returns a nonzero value. If the environment was successfully established, the **fesetenv** subroutine returns zero. Otherwise, it returns a nonzero value.

Related Information

"feholdexcept Subroutine" on page 225 and "feupdateenv Subroutine" on page 231

fegetexceptflag or fesetexceptflag Subroutine

Purpose

Gets and sets floating-point status flags.

Syntax

```
#include <fenv.h>
int fegetexceptflag (flagp, excepts)
feexcept_t *flagp;
int excepts;
int fesetexceptflag (flagp, excepts)
const fexcept t *flagp;
int excepts;
```

Description

The fegetexceptflag subroutine attempts to store an implementation-defined representation of the states of the floating-point status flags indicated by the excepts parameter in the object pointed to by the flagp parameter.

The **fesetexceptflag** subroutine attempts to set the floating-point status flags indicated by the *excepts* parameter to the states stored in the object pointed to by the *flagp* parameter. The value pointed to by the flagp parameter shall have been set by a previous call to the fegetexceptflag subroutine whose second argument represented at least those floating-point exceptions represented by the excepts parameter. This subroutine does not raise floating-point exceptions. It only sets the state of the flags.

Parameters

flagp Points to the object that holds the implementation-defined representation of the states of the

floating-point status flags.

Points to an implementation-defined representation of the states of the floating-point status flags. excepts

Return Values

If the representation was successfully stored, the fegetexceptflag parameter returns zero. Otherwise, it returns a nonzero value. If the excepts parameter is zero or if all the specified exceptions were successfully set, the fesetexceptflag subroutine returns zero. Otherwise, it returns a nonzero value.

Related Information

"feraiseexcept Subroutine" on page 228 and "fetestexcept Subroutine" on page 231.

fegetround or fesetround Subroutine

Purpose

Gets and sets the current rounding direction.

Syntax

```
#include <fenv.h>
int fegetround (void)
int fesetround (round)
int round;
```

Description

The **fegetround** subroutine gets the current rounding direction.

The **fesetround** subroutine establishes the rounding direction represented by the *round* parameter. If the round parameter is not equal to the value of a rounding direction macro, the rounding direction is not changed.

Parameters

round

Specifies the rounding direction.

Return Values

The fegetround subroutine returns the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

The **fesetround** subroutine returns a zero value if the requested rounding direction was established.

feholdexcept Subroutine

Purpose

Saves current floating-point environment.

Syntax

```
#include <fenv.h>
int feholdexcept (envp)
fenv_t *envp;
```

Description

The **feholdexcept** subroutine saves the current floating-point environment in the object pointed to by *envp*, clears the floating-point status flags, and installs a non-stop (continue on floating-point exceptions) mode for all floating-point exceptions.

Parameters

envp

Points to the current floating-point environment.

Return Values

The **feholdexcept** subroutine returns zero if non-stop floating-point exception handling was successfully installed.

Related Information

The "feupdateenv Subroutine" on page 231.

fence Subroutine

Purpose

Allows you to request and change the virtual shared disk fence map.

Syntax

```
#include <vsd ioctl.h>
int ioctl(FileDescriptor, Command, Argument)
int FileDescriptor, Command;
void *Argument;
```

Description

Use this subroutine to request and change the virtual shared disk fence map. The fence map, which controls whether virtual shared disks can send or satisfy requests from virtual shared disks at remote nodes, is defined as:

```
struct vsd FenceMap
                      /* This is the argument to the VSD fence ioctl. */
  ulong
  vsd minorBitmap t minornoBitmap; /* Bitmap of minor numbers to fence
                                          (supports 10000 vsds)
  vsd Fence Bitmap t
                        nodesBitmap;
                                      /* Nodes to (un)fence these vsds from
                                          (supports node numbers 1-2048)
}vsd_FenceMap_t
```

The flags VSD_FENCE and VSD_UNFENCE are mutually exclusive — an ioctl can either fence a set of virtual shared disks or unfence a set of virtual shared disks, but not both. The minornoBitmap denotes which virtual shared disks are to be fenced/unfenced from the nodes specified in the *nodesBitmap*.

Parameters

FileDescriptor Specifies the open file descriptor for which the control operation is to be performed.

Command Specifies the control function to be performed. The value of this parameter is always

GIOCFENCE.

Argument Specifies a pointer to a **vsd_fence_map** structure.

The *flags* field of the **vsd_fence_map** structure determines the type of operation that is performed. The flags could be set with one or more options using the OR operator. These options are as follows:

VSD_FENCE_FORCE If this option is specified, a node can unfence itself.

VSD_FENCE_GET Denotes a query request. **VSD FENCE** Denotes a fence request. VSD_UNFENCE Denotes an unfence request.

Examples

The following example fences a virtual shared disk with a minor number of 7 from node 4 and 5, and unfences a virtual shared disk with a minor number of 5 from node 1:

```
int fd:
vsd FenceMap t FenceMap;
/* Clear the FenceMap */
bzero(FenceMap, sizeof(vsd FenceMap t));
/* fence nodes 4,5 from minor 7 */
FenceMap.flags = VSD FENCE;
MAP SET(7, FenceMap.minornoBitmap);
MAP_SET(4, FenceMap.nodesBitmap);
MAP SET(5, FenceMap.nodesBitmap);
/* Issue the fence request */
ioctl(fd,GIOCFENCE,&FenceMap);
/* Unfence node 1 from minor 5*/
bzero(FenceMap, sizeof(vsd FenceMap t));
FenceMap.flags = VSD UNFENCE | VSD FENCE FORCE;
MAP SET(5, FenceMap.minornoBitmap);
MAP SET(1, FenceMap.nodesBitmap);
/* Issue the fence request */
ioctl(fd,GIOCFENCE,&FenceMap);
```

Return Values

If the request succeeds, the ioctl returns 0. In the case of an error, a value of -1 is returned with the global variable errno set to identify the error.

Error Values

The fence ioctl subroutine can return the following error codes:

EACCES Indicates that an unfence was requested from a fenced node without the

VSD FENCE FORCE option.

EINVAL Indicates an invalid request (ambiguous flags or unidentified virtual shared

disks).

ENOCONNECT Indicates that either the primary or the secondary node for a virtual shared

disk to be fenced is not a member of the virtual shared disk group, or the

virtual shared disk in question is in the **stopped** state.

ENOTREADY Indicates that the group is not active or the Recoverable virtual shared

disk subsystem is not available.

ENXIO Indicates that the Virtual shared disk driver is being unloaded.

feof, ferror, clearerr, or fileno Macro

Purpose

Checks the status of a stream.

Library

Standard C Library (libc.a)

Syntax

int feof (Stream) **FILE** *Stream; int ferror (Stream) FILE *Stream; void clearerr (Stream) FILE *Stream;

#include <stdio.h>

int fileno (Stream) **FILE** *Stream;

Description

The feof macro inquires about the end-of-file character (EOF). If EOF has previously been detected reading the input stream specified by the Stream parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **ferror** macro inquires about input or output errors. If an I/O error has previously occurred when reading from or writing to the stream specified by the Stream parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The clearerr macro inquires about the status of a stream. The clearerr macro resets the error indicator and the EOF indicator to a value of 0 for the stream specified by the Stream parameter.

The fileno macro inquires about the status of a stream. The fileno macro returns the integer file descriptor associated with the stream pointed to by the Stream parameter. Otherwise a value of -1 is returned.

Parameters

Stream

Specifies the input or output stream.

Related Information

The **fopen**, **freopen**, or **fdopen** ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, open ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine.

Input and Output Handling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

feraiseexcept Subroutine

Purpose

Raises the floating-point exception.

Syntax

#include <fenv.h>

int feraiseexcept (excepts) int excepts;

Description

The feraiseexcept subroutine attempts to raise the supported floating-point exceptions represented by the excepts parameter. The order in which these floating-point exceptions are raised is unspecified.

Parameters

excepts

Points to the floating-point exceptions.

Return Values

If the argument is zero or if all the specified exceptions were successfully raised, the feraiseexcept subroutine returns a zero. Otherwise, it returns a nonzero value.

Related Information

"feclearexcept Subroutine" on page 223, "fegetexceptflag or fesetexceptflag Subroutine" on page 224, "fetestexcept Subroutine" on page 231.

fetch_and_add Subroutine

Purpose

Updates a single word variable atomically.

Library

Standard C library (libc.a)

Syntax

```
#include <sys/atomic op.h>
int fetch and add (word addr, value)
atomic p word addr;
int value:
```

Description

The fetch_and_add subroutine increments one word in a single atomic operation. This operation is useful when a counter variable is shared between several threads or processes. When updating such a counter variable, it is important to make sure that the fetch, update, and store operations occur atomically (are not interruptible). For example, consider the sequence of events which could occur if the operations were interruptible:

- 1. A process fetches the counter value and adds one to it.
- 2. A second process fetches the counter value, adds one, and stores it.
- 3. The first process stores its value.

The result of this is that the update made by the second process is lost.

Traditionally, atomic access to a shared variable would be controlled by a mechanism such as semaphores. Compared to such mechanisms, the **fetch and add** subroutine requires very little overhead, and provided that the counter variable fits in a single machine word, this subroutine provides a highly efficient way of performing this operation.

Note: The word containing the counter variable must be aligned on a full word boundary.

Parameters

word addr Specifies the address of the word variable to be incremented. Specifies the value to be added to the word variable. value

Return Values

This subroutine returns the original value of the word.

Related Information

The fetch and and ("fetch and or fetch and or Subroutine") subroutine, fetch and or ("fetch_and_and or fetch_and_or Subroutine") subroutine, compare_and_swap ("compare_and_swap Subroutine" on page 143) subroutine.

fetch and and or fetch and or Subroutine

Purpose

Sets or clears bits in a single word variable atomically.

Library

Standard C library (libc.a)

Syntax

```
#include <sys/atomic op.h>
uint fetch and and ( word addr, mask)
atomic_p word_addr;
int mask;
uint fetch and or ( word addr, mask)
atomic p word addr;
int mask;
```

Description

The **fetch and and and fetch and or** subroutines respectively clear and set bits in one word, according to a bit mask, in a single atomic operation. The **fetch_and_and** subroutine clears bits in the word which correspond to clear bits in the bit mask, and the fetch_and_or subroutine sets bits in the word which correspond to set bits in the bit mask.

These operations are useful when a variable containing bit flags is shared between several threads or processes. When updating such a variable, it is important that the fetch, bit clear or set, and store operations occur atomically (are not interruptible). For example, consider the sequence of events which could occur if the operations were interruptible:

- 1. A process fetches the flags variable and sets a bit in it.
- 2. A second process fetches the flags variable, sets a different bit, and stores it.
- 3. The first process stores its value.

The result is that the update made by the second process is lost.

Traditionally, atomic access to a shared variable would be controlled by a mechanism such as semaphores. Compared to such mechanisms, the fetch_and_and and fetch_and_or subroutines require very little overhead, and provided that the flags variable fits in a single machine word, they provide a highly efficient way of performing this operation.

Note: The word containing the flag bits must be aligned on a full word boundary.

Parameters

word_addr mask

Specifies the address of the single word variable whose bits are to be cleared or set.

Specifies the bit mask which is to be applied to the single word variable.

Return Values

These subroutines return the original value of the word.

Related Information

The fetch and add ("fetch and add Subroutine" on page 229) subroutine, compare and swap ("compare_and_swap Subroutine" on page 143) subroutine.

fetestexcept Subroutine

Purpose

Tests floating-point exception flags.

Syntax

#include <fenv.h>

int fetestexcept (excepts) int excepts;

Description

The fetestexcept subroutine determines which of a specified subset of the floating-point exception flags are currently set. The excepts parameter specifies the floating-point status flags to be gueried.

Parameters

excepts

Specifies the floating-point status flags to be queried.

Return Values

The fetestexcept subroutine returns the value of the bitwise-inclusive OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in excepts.

Related Information

"feclearexcept Subroutine" on page 223, "fegetexceptflag or fesetexceptflag Subroutine" on page 224, and "feraiseexcept Subroutine" on page 228

feupdateenv Subroutine

Purpose

Updates floating-point environment.

Syntax

#include <fenv.h>

```
int feupdateenv (envp)
const fenv_t *envp;
```

Description

The **feupdateenv** subroutine attempts to save the currently raised floating-point exceptions in its automatic storage, attempts to install the floating-point environment represented by the object pointed to by the *envp* parameter, and attempts to raise the saved floating-point exceptions. The *envp* parameter point to an object set by a call to **feholdexcept** or **fegetenv**, or equal a floating-point environment macro.

Parameters

envp

Points to an object set by a call to the **feholdexcept** or the **fegetenv** subroutine, or equal a floating-point environment macro.

Return Values

The feupdateenv subroutine returns a zero value if all the required actions were successfully carried out.

Related Information

"fegetenv or fesetenv Subroutine" on page 223 and "feholdexcept Subroutine" on page 225.

finfo or ffinfo Subroutine

Purpose

Returns file information.

Library

Standard C library (libc.a)

Syntax

```
#include <sys/finfo.h>
int finfo(Path1, cmd, buffer, length)
const char *Path1;
int cmd;
void *buffer;
int length;
int ffinfo (fd, cmd, buffer, length)
int fd;
int cmd;
void *buffer;
int length;
```

Description

The finfo and ffinfo subroutines return specific file information for the specified file.

Parameters

Path1 Path name of a file system object to query.

fd File descriptor for an open file to query.

cmd Specifies the type of file information to be returned.

buffer User supplied buffer which contains the file information upon successful return.

/usr/include/sys/finfo.h describes the buffer.

length Length of the query buffer.

Commands

F_PATHCONF When the F_PATHCONF command is specified, a file's

implementation information is returned.

Note: The operating system provides another subroutine that retrieves file implementation characteristics, pathconf

("pathconf or fpathconf Subroutine" on page 774) command. While the finfo and ffinfo subroutines can be used to retrieve file information, it is preferred that

programs use the pathconf interface.

F DIOCAP When the F_DIOCAP command is specified, the file's

direct 1/0 capability information is returned. The buffer supplied by the application is of type struct diocapbuf *.

Return Values

Upon successful completion, the finfo and ffinfo subroutines return a value of 0 and the user supplied buffer is correctly filled in with the file information requested. If the finfo or ffinfo subroutines were unsuccessful, a value of -1 is returned and the global errno variable is set to indicate the error.

Error Codes

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

EINVAL If the length specified for the user buffer is greater than

MAX_FINFO_BUF.

If the command argument is not supported. If **F_DIOCAP** command is

specified and the file object does not support Direct I/O.

ENAMETOOLONG The length of the Path parameter string exceeds the PATH_MAX

value.

ENOENT The named file does not exist or the Path parameter points to an

empty string.

ENOTDIR A component of the path prefix is not a directory.

EBADF File descriptor provided is not valid.

Related Information

The **pathconf** ("pathconf or fpathconf Subroutine" on page 774) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

flockfile, ftrylockfile, funlockfile Subroutine

Purpose

Provides for explicit application-level locking of stdio (FILE*) objects.

Library

Standard Library (libc.a)

Syntax

#include <stdio.h> void flockfile (FILE * file) int ftrylockfile (FILE * file) void funlockfile (FILE * file)

Description

The **flockfile**, **ftrylockfile** and **funlockfile** functions provide for explicit application-level locking of stdio (FILE*) objects. These functions can be used by a thread to delineate a sequence of I/O statements that are to be executed as a unit.

The **flockfile** function is used by a thread to acquire ownership of a (**FILE***) object.

The ftrylockfile function is used by a thread to acquire ownership of a (FILE*) object if the object is available; ftrylockfile is a non-blocking version of flockfile.

The **funlockfile** function is used to relinquish the ownership granted to the thread. The behavior is undefined if a thread other than the current owner calls the funlockfile function.

Logically, there is a lock count associated with each (FILE*) object. This count is implicitly initialised to zero when the (FILE*) object is created. The (FILE*) object is unlocked when the count is zero. When the count is positive, a single thread owns the (FILE*) object. When the flockfile function is called, if the count is zero or if the count is positive and the caller owns the (FILE*) object, the count is incremented. Otherwise, the calling thread is suspended, waiting for the count to return to zero. Each call to funlockfile decrements the count. This allows matching calls to flockfile (or successful calls to ftrylockfile) and funlockfile to be nested.

All functions that reference (FILE*) objects behave as if they use flockfile and funlockfile internally to obtain ownership of these (FILE*) objects.

Return Values

None for flockfile and funlockfile. The function ftrylock returns zero for success and non-zero to indicate that the lock cannot be acquired.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) subroutines.

Realtime applications may encounter priority inversion when using FILE locks. The problem occurs when a high priority thread locks a file that is about to be unlocked by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of 7434 ways, such as by having critical sections that are guarded by file locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

Related Information

The getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked ("getc_unlocked, getchar unlocked, putc unlocked, putchar unlocked Subroutines" on page 302) subroutine.

floor, floorf, floorl, nearest, trunc, itrunc, or uitrunc Subroutine

Purpose

The floor subroutine, floorf subroutine, floorl subroutine, nearest subroutine, and trunc subroutine, round floating-point numbers to floating-point integer values.

The itrunc subroutine and uitrunc subroutine round floating-point numbers to signed and unsigned integers, respectively.

Libraries

```
IEEE Math Library (libm.a)
or System V Math Library (libmsaa.a)
Standard C Library (libc.a) (separate syntax follows)
```

Syntax

```
#include <math.h>
double floor (x)
double x;
float floorf (x)
float x;
long double floor (x)
long double x;
double nearest (x)
double x;
double trunc (x)
double x;
Standard C Library (libc.a)
#include <stdlib.h>
#include <limits.h>
int itrunc (x)
double x;
unsigned int uitrunc (x)
double x;
```

Description

The floor subroutine and floorI subroutines return the largest floating-point integer value not greater than the x parameter.

An application wishing to check for error situations should set errno to zero and call feclearexcept(FE ALL EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

The **nearest** subroutine returns the nearest floating-point integer value to the x parameter. If x lies exactly halfway between the two nearest floating-point integer values, an even floating-point integer is returned.

The **trunc** subroutine returns the nearest floating-point integer value to the x parameter in the direction of 0. This is equivalent to truncating off the fraction bits of the x parameter.

Note: The default floating-point rounding mode is round to nearest. All C main programs begin with the rounding mode set to round to nearest.

The **itrunc** subroutine returns the nearest signed integer to the x parameter in the direction of 0. This is equivalent to truncating the fraction bits from the x parameter and then converting x to a signed integer.

The **uitrunc** subroutine returns the nearest unsigned integer to the x parameter in the direction of 0. This action is equivalent to truncating off the fraction bits of the x parameter and then converting x to an unsigned integer.

Note: Compile any routine that uses subroutines from the libm.a library with the -Im flag. To compile the floor.c file, for example, enter:

cc floor.c -lm

The **itrunc**, **uitrunc**, **trunc**, and **nearest** subroutines are not part of the ANSI C Library.

Parameters

- Specifies a double-precision floating-point value. For the floorl subroutine, specifies a long double-precision floating-point value.
- Specifies a double-precision floating-point value. For the floorl subroutine, specifies some long У double-precision floating-point value.

Return Values

Upon successful completion, the floor, floorf, and floorl subroutine returns the largest integral value not greater than x, expressed as a double, float, or long double, as appropriate for the return type of the function.

If x is NaN, a NaN is returned.

If x is ± 0 or $\pm \ln x$ is returned.

If the correct value would cause overflow, a range error occurs and the floor, floorf and floorI subroutines return the value of the macro -HUGE VAL. -HUGE VALF and -HUGE VALL, respectively.

Error Codes

The **itrunc** and **uitrunc** subroutines return the **INT MAX** value if x is greater than or equal to the **INT_MAX** value and the **INT_MIN** value if x is equal to or less than the **INT_MIN** value. The **itrunc** subroutine returns the **INT MIN** value if x is a Quiet NaN(not-a-number) or Silent NaN. The **uitrunc** subroutine returns 0 if x is a Quiet NaN or Silent NaN. (The INT_MAX and INT_MIN values are defined in the **limits.h** file.) The **uitrunc** subroutine **INT MAX** if x is greater than **INT MAX** and 0 if x is less than or equal 0.0

Files

float.h Contains the ANSI C FLT_ROUNDS macro.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, _class, finite, isnan, or unordered Subroutines" on page 138.

The **fp read rnd** or **fp swap rnd** ("fp read rnd or fp swap rnd Subroutine" on page 259) subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

128-Bit long double Floating-Point Format in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

math.h in AIX 5L Version 5.2 Files Reference.

fma, fmaf, or fmal Subroutine

Purpose

Floating-point multiply-add.

Syntax

```
#include <math.h>
double fma (x, y, z)
double x;
double y;
double z;
float fmaf (x, y, z)
float x;
float y;
float z;
long double fmal (x, y, z)
long double x;
long double y;
long double z;
```

Description

The **fma**, **fmaf**, and **fmal** subroutines compute (x * y) + z, rounded as one ternary operation. They compute the value (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of FLT ROUNDS.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE ALL EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

- Specifies the value to be multiplied by the *y* parameter. X
- Specifies the value to be multiplied by the *x* parameter.
- Specifies the value to be added to the product of the *x* and *y* parameters. 7

Return Values

Upon successful completion, the **fma**, **fmaf**, and **fmal** subroutines return (x * y) + z, rounded as one ternary operation.

If x or y are NaN, a NaN is returned.

If x multiplied by y is an exact infinity and z is also an infinity but with the opposite sign, a domain error occurs, and a NaN is returned.

If one of the x and y parameters is infinite, the other is zero, and the z parameter is not a NaN, a domain error occurs, and a NaN is returned.

If one of the x and y parameters is infinite, the other is zero, and z is a NaN, a NaN is returned and a domain error may occur.

If x^*y is not 0*Inf nor Inf*0 and z is a NaN, a NaN is returned.

Related Information

"feclearexcept Subroutine" on page 223 and "fetestexcept Subroutine" on page 231

math.h in AIX 5L Version 5.2 Files Reference.

fmax, fmaxf, or fmaxl Subroutine

Purpose

Determines the maximum numeric value of two floating-point numbers.

Syntax

```
#include <math.h>
double fmax (x, y)
double x;
double y;

float fmaxf (x, y)
float x;
float y;

long double fmaxl (x, y)
long double x;
long double y;
```

Description

The **fmax**, **fmaxf**, and **fmaxl** subroutines determine the maximum numeric value of their arguments. NaN arguments are treated as missing data. If one argument is a NaN and the other numeric, the **fmax**, **fmaxf**, and **fmaxl** subroutines choose the numeric value.

Parameters

```
x Specifies the value to be computed.y Specifies the value to be computed.
```

Return Values

Upon successful completion, the **fmax**, **fmaxf**, and **fmaxl** subroutines return the maximum numeric value of their arguments.

If one argument is a NaN, the other argument is returned.

If x and y are NaN, a NaN is returned.

Related Information

"fdim, fdimf, or fdiml Subroutine" on page 222 and "madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout, omout, fmout, m_out, sdiv, or itom Subroutine" on page 661

math.h in AIX 5L Version 5.2 Files Reference.

fminf or fminl Subroutine

Purpose

Determines the minimum numeric value of two floating-point numbers.

Syntax

```
#include <math.h>
float fminf (x, y)
float x;
float y;
long double fminl (x, y)
long double x;
long double y;
```

Description

The fminf and fminI subroutines determine the minimum numeric value of their arguments. NaN arguments are treated as missing data If one argument is a NaN and the other numeric, the fminf and fminI subroutines choose the numeric value.

Parameters

```
Specifies the value to be computed.
Specifies the value to be computed.
```

Return Values

Upon successful completion, the fminf and fminl subroutines return the minimum numeric value of their arguments.

If one argument is a NaN, the other argument is returned.

If x and y are NaN, a NaN is returned.

Related Information

"fdim, fdimf, or fdiml Subroutine" on page 222, "fmax, fmaxf, or fmaxl Subroutine" on page 238.

math.h in AIX 5L Version 5.2 Files Reference.

fmod, fmodf, or fmodl Subroutine

Purpose

Computes the floating-point remainder value.

Syntax

```
#include <math.h>
float fmodf (x, y)
float x;
float y;
long double fmod1 (x)
```

```
long double x, y;
double fmod (x, y)
double x, y;
```

Description

The **fmodf**, **fmodl**, and **fmod** subroutines return the floating-point remainder of the division of x by y.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

x Specifies the value to be computed.y Specifies the value to be computed.

Return Values

The **fmodf**, **fmodl**, and **fmod** subroutines return the value x- i * y, for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 is returned.

If x or y is NaN, a NaN is returned.

If y is zero, a domain error occurs, and a NaN is returned.

If x is infinite, a domain error occurs, and a NaN is returned.

If x is ± 0 and y is not zero, ± 0 is returned.

If x is not infinite and y is $\pm \ln f$, x is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, _class, finite, isnan, or unordered Subroutines" on page 138.

math.h in AIX 5L Version 5.2 Files Reference.

fmtmsg Subroutine

Purpose

Display a message in the specified format on standard error, the console, or both.

Library

Standard C Library (libc.a)

Syntax

```
#include <fmtmsg.h>
int fmtmsg (long Classification,
const char *Label,
int Severity,
cont char *Text;
cont char *Action,
cont char *Tag)
```

Description

The fmtmsg subroutine can be used to display messages in a specified format instead of the traditional **printf** subroutine interface.

Base on a message's classification component, the **fmtmsg** subroutine either writes a formatted message to standard error, the console, or both.

A formatted message consists of up to five parameters. The Classification parameter is not part of a message displayed to the user, but defines the source of the message and directs the display of the formatted message.

Parameters

Classification

Contains identifiers from the following groups of major classifications and subclassifications. Any one identifier from a subclass may be used in combination with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both standard error and system console).

major classifications

Identifies the source of the condition. Identifiers are: MM_HARD (hardware), MM_SOFT (software), and MM_FIRM (firmware).

message source subclassifications

Identifies the type of software in which the problem is detected. Identifiers are: MM_APPL (application), MM_UTIL (utility), and MM_OPSYS (operating system).

display subclassification

Indicates where the message is to be displayed. Identifiers are: MM_PRINT to display the message on the standard error stream, MM CONSOLE to display the message on the system console. One or both identifiers may be used.

status subclassifications

Indicates whether the application will recover from the condition. Identifiers are:MM_RECOVER (recoverable) and MM_RECOV (non-recoverable).

An additional identifier, MM_NULLMC, identifies that no classification component is supplied for the message.

Identifies the source to the message. The format is two fields separated by a colon. The

first field is up to 10 bytes, the second field is up to 14 bytes.

Severity

Label

Text Describes the error condition that produced the message. The character string is not

limited to a specific size. If the character string is null then a message will be issued

stating that no text has been provided.

Action Describes the first step to be taken in the error-recovery process. The fmtmsg subroutine

precedes the action string with the prefix: T0 FIX:. The Action string is not limited to a

specific size.

Tag An identifier which references online documentation for the message. Suggested usage is

that tag includes the Label and a unique identifying number. A sample tag is UX:cat:146.

Environment Variables

The MSGVERB (message verbosity) environment variable controls the behavior of the fmtmsg subroutine.

MSGVERB tells the fmtmsg subroutine which message components it is to select when writing messages to standard error. The value of MSGVERB is a colon-separated list of optional keywords. MSGVERB can be set as follows:

```
MSGVERB=[keyword[:keyword[:...]]]
export MSGVERB
```

Valid keywords are: Label, Severity, Text, Action, and Tag. If MSGVERB contains a keyword for a component and the component's value is not the component's null value, **fmtmsq** subroutine includes that component in the message when writing the message to standard error. If MSGVERB does not include a keyword for a message component, that component is not included in the display of the message. The keywords may appear in any order. If MSGVERB is not defined, if its value is the null string, if its value is not of the correct format, of if it contains keywords other than the valid ones listed previously, the fmtmsg subroutine selects all components.

MSGVERB affects only which components are selected for display to standard error. All message components are included in console messages.

Application Usage

One or more message components may be systematically omitted from messages generated by an application by using the null value of the parameter for that component. The table below indicates the null values and identifiers for **fmtmsg** subroutine parameters.

Parameter	Туре	Null-Value	Identifier
label	char*	(char*)0	MM_NULLLBL
severity	int	0	MM_NULLSEV
class	long	0L	MM_NULLMC
text	char*	(char*)0	MM_NULLTXT
action	char*	(char*)0	MM_NULLACT
tag	char*	(char*)0	MM_NULLTAG

Another means of systematically omitting a component is by omitting the component keywords when defining the MSGVERB environment variable.

Return Values

The exit codes for the **fmtmsg** subroutine are the following:

MM_OK The function succeeded. MM_NOTOK The function failed completely.

MM_MOMSG The function was unable to generate a message on standard error.

MM_NOCON The function was unable to generate a console message.

Examples

1. The following example of the **fmtmsg** subroutine:

```
fmtmsg(MM PRINT, "UX:cat", MM ERROR, "illegal option",
"refer tp cat in user's reference manual", "UX:cat:001")
```

produces a complete message in the specified message format:

```
UX:cat ERROR: illegal option
TO FIX: refer to cat in user's reference manual UX:cat:001
```

2. When the environment variable MSGVERB is set as follows:

```
MSGVERB=severity:text:action
```

and the Example 1 is used, the **fmtmsg** subroutine produces:

```
ERROR: illegal option
```

TO FIX: refer to cat in user's reference manual UX:cat:001

Related Information

The printf ("printf, fprintf, sprintf, sprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine.

fnmatch Subroutine

Purpose

Matches file name patterns.

Library

Standard C Library (libc. a)

Syntax

```
#include <fnmatch.h>
int fnmatch ( Pattern, String, Flags);
int Flags;
const char *Pattern, *String;
```

Description

The **fnmatch** subroutine checks the string specified by the *String* parameter to see if it matches the pattern specified by the Pattern parameter.

The fnmatch subroutine can be used by an application or command that needs to read a dictionary and apply a pattern against each entry; the **find** command is an example of this. It can also be used by the pax command to process its Pattern variables, or by applications that need to match strings in a similar manner.

Parameters

Pattern

Contains the pattern to which the String parameter is to be compared. The Pattern parameter can include the following special characters:

* (asterisk)

Matches zero, one, or more characters.

? (question mark)

Matches any single character, but will not match 0 (zero) characters.

[] (brackets)

Matches any one of the characters enclosed within the brackets. If a pair of characters separated by a dash are contained within the brackets, the pattern matches any character that lexically falls between the two characters in the current locale.

String

Contains the string to be compared against the *Pattern* parameter.

Flags

Contains a bit flag specifying the configurable attributes of the comparison to be performed by the **fnmatch** subroutine.

The Flags parameter modifies the interpretation of the Pattern and String parameters. It is the bitwise inclusive OR of zero or more of the following flags (defined in the fnmatch.h file):

FNM PATHNAME

Indicates the / (slash) in the String parameter matches a / in the Pattern parameter.

FNM PERIOD

Indicates a leading period in the String parameter matches a period in the Pattern parameter.

FNM NOESCAPE

Enables quoting of special characters using the \ (backslash).

Ignores uppercase and lowercase when matching alphabetic characters (available only in AIX 5.1 or later).

If the **FNM PATHNAME** flag is set in the *Flags* parameter, a / (slash) in the *String* parameter is explicitly matched by a / in the *Pattern* parameter. It is not matched by either the * (asterisk) or ? (question-mark) special characters, nor by a bracket expression. If the FNM PATHNAME flag is not set, the / is treated as an ordinary character.

If the FNM_PERIOD flag is set in the Flags parameter, then a leading period in the String parameter only matches a period in the *Pattern* parameter; it is not matched by either the asterisk or question-mark special characters, nor by a bracket expression. The setting of the FNM PATHNAME flag determines a period to be leading, according to the following rules:

- If the FNM_PATHNAME flag is set, a . (period) is leading only if it is the first character in the String parameter or if it immediately follows a /.
- If the FNM PATHNAME flag is not set, a . (period) is leading only if it is the first character of the String parameter. If FNM_PERIOD is not set, no special restrictions are placed on matching a period.

If the FNM_NOESCAPE flag is not set in the Flags parameter, a \ (backslash) character in the Pattern parameter, followed by any other character, will match that second character in the String parameter. For example, \\ will match a backslash in the String parameter. If the FNM_NOESCAPE flag is set, a \ (backslash) will be treated as an ordinary character.

Return Values

If the value in the String parameter matches the pattern specified by the Pattern parameter, the fnmatch subroutine returns 0. If there is no match, the **fnmatch** subroutine returns the **FNM_NOMATCH** constant, which is defined in the **fnmatch.h** file. If an error occurs, the **fnmatch** subroutine returns a nonzero value.

Files

/usr/include/fnmatch.h

Contains system-defined flags and constants.

Related Information

The **glob** ("glob Subroutine" on page 401) subroutine, **globfree** ("globfree Subroutine" on page 404) subroutine, regcomp subroutine, regfree subroutine, regerror subroutine, regexec subroutine.

The **find** command, **pax** command.

Files, Directories, and File Systems for Programmers and Understanding Internationalized Regular Expression Subroutines Ln AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging **Programs**

fopen, fopen64, freopen, freopen64 or fdopen Subroutine

Purpose

Opens a stream.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdio.h>
FILE *fopen ( Path, Type)
const char *Path, *Type;
FILE *fopen64 ( Path, Type)
char *Path, *Type;
FILE *freopen (Path, Type, Stream)
const char *Path, *Type;
FILE *Stream;
FILE *freopen64 (Path, Type, Stream)
char *Path, *Type;
FILE *Stream;
FILE *fdopen ( FileDescriptor, Type)
int FileDescriptor;
const char *Type;
```

Description

The **fopen** and **fopen64** subroutines open the file named by the *Path* parameter and associate a stream with it and return a pointer to the **FILE** structure of this stream.

When you open a file for update, you can perform both input and output operations on the resulting stream. However, an output operation cannot be directly followed by an input operation without an intervening fflush subroutine call or a file positioning operation (fseek, fseeko, fseeko64, fsetpos, fsetpos64 or rewind subroutine). Also, an input operation cannot be directly followed by an output operation without an intervening flush or file positioning operation, unless the input operation encounters the end of the file.

When you open a file for appending (that is, when the Type parameter is set to a), it is impossible to overwrite information already in the file.

If two separate processes open the same file for append, each process can write freely to the file without destroying the output being written by the other. The output from the two processes is intermixed in the order in which it is written to the file.

Note: If the data is buffered, it is not actually written until it is flushed.

The **freopen** and **freopen64** subroutines first attempt to flush the stream and close any file descriptor associated with the Stream parameter. Failure to flush the stream or close the file descriptor is ignored.

The freopen and freopen64 subroutines substitute the named file in place of the open stream. The original stream is closed regardless of whether the subsequent open succeeds. The freopen and freopen64 subroutines returns a pointer to the FILE structure associated with the Stream parameter. The freopen and freopen64 subroutines is typically used to attach the pre-opened streams associated with standard input (stdin), standard output (stdout), and standard error (stderr) streams to other files.

The **fdopen** subroutine associates a stream with a file descriptor obtained from an **openx** subroutine, **dup** subroutine, creat subroutine, or pipe subroutine. These subroutines open files but do not return pointers to FILE structures. Many of the standard I/O package subroutines require pointers to FILE structures.

The Type parameter for the **fdopen** subroutine specifies the mode of the stream, such as r to open a file for reading, or a to open a file for appending (writing at the end of the file). The mode value of the Type parameter specified with the **fdopen** subroutine must agree with the mode of the file specified when the file was originally opened or created.

The largest value that can be represented correctly in an object of type off t will be established as the offset maximum in the open file description.

Parameters

Path Туре

Points to a character string that contains the name of the file to be opened. Points to a character string that has one of the following values:

- Opens a text file for reading.
- Creates a new text file for writing, or opens and truncates a file to 0 length. w
- Appends (opens a text file for writing at the end of the file, or creates a file for а writing).
- rb Opens a binary file for reading.
- wb Creates a binary file for writing, or opens and truncates a file to 0.
- Appends (opens a binary file for writing at the end of the file, or creates a file for writing).
- Opens a file for update (reading and writing). r+
- Truncates or creates a file for update. W+
- a± Appends (opens a text file for writing at end of file, or creates a file for writing).

r+b, rb+

Opens a binary file for update (reading and writing).

w+b, wb+

Creates a binary file for update, or opens and truncates a file to 0 length.

Appends (opens a binary file for update, writing at the end of the file, or creates a file for writing).

Note: The operating system does not distinguish between text and binary files. The b value in the *Type* parameter value is ignored.

Specifies the input stream.

FileDescriptor Specifies a valid open file descriptor.

Stream

Return Values

If the **fdopen**, **fopen**, **fopen64**, **freopen** or **freopen64** subroutine is unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The fopen, fopen64, freopen and freopen64 subroutines are unsuccessful if the following is true:

EACCES Search permission is denied on a component of the path prefix, the file exists and the

permissions specified by the mode are denied, or the file does not exist and write permission

is denied for the parent directory of the file to be created.

ELOOP Too many symbolic links were encountered in resolving path.

EINTR A signal was received during the process.

EISDIR The named file is a directory and the process does not have write access to it.

ENAMETOOLONG The length of the filename exceeds **PATH_MAX** or a pathname component is longer than

NAME_MAX.

ENFILE The maximum number of files allowed are currently open.

ENOENTThe named file does not exist or the *File Descriptor* parameter points to an empty string. **ENOSPC**The file is not yet created and the directory or file system to contain the new file cannot be

expanded.

ENOTDIR A component of the path prefix is not a directory.

ENXIO The named file is a character- or block-special file, and the device associated with this

special file does not exist.

EOVERFLOW The named file is a regular file and the size of the file cannot be represented correctly in an

object of type off_t.

EROFS The named file resides on a read-only file system and does not have write access.

ETXTBSY The file is a pure-procedure (shared-text) file that is being executed and the process does not

have write access.

The **fdopen**, **fopen**, **fopen64**, **freopen** and **freopen64** subroutines are unsuccessful if the following is true:

EINVAL The value of the *Type* argument is not valid.

EINVAL The value of the *mode* argument is not valid.

EMFILE FOPEN_MAX streams are currently open in the calling process. **EMFILE STREAM_MAX** streams are currently open in the calling process.

ENAMETOOLONG Pathname resolution of a symbolic link produced an intermediate result whose length

exceeds PATH_MAX.

ENOMEM Insufficient storage space is available.

The **freopen** and **fopen** subroutines are unsuccessful if the following is true:

EOVERFLOW The named file is a size larger than 2 Gigabytes.

The **fdopen** subroutine is unsuccessful if the following is true:

EBADF The value of the *File Descriptor* parameter is not valid.

POSIX

Truncates to 0 length or creates text file for writing.
 Truncates to 0 length or creates text file for update.
 Opens or creates text file for writing at end of file.

a+ Opens or creates text file for update, writing at end of file.

SAA

At least eight streams, including three standard text streams, can open simultaneously. Both binary and text modes are supported.

Related Information

The fclose or fflush ("fclose or fflush Subroutine" on page 213) subroutine, fseek, fseeko, fseeko64, rewind, ftell, ftello, ftello64, fgetpos, fgetpos64 or fsetpos ("fseek, fseeko, fseeko64, rewind, ftell, ftello, ftello64, fgetpos, fgetpos64, fsetpos, or fsetpos64 Subroutine" on page 273) subroutine, open, open64, openx, or creat ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine, setbuf, setbuffer, or setlinebuf subroutine.

The Input and Output Handling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fork, f_fork, or vfork Subroutine

Purpose

Creates a new process.

Libraries

fork, f_fork, and vfork: Standard C Library (libc.a)

Syntax

#include <unistd.h>
pid_t fork(void)
pid_t f_fork(void)
int vfork(void)

Description

The **fork** subroutine creates a new process. The new process (child process) is an almost exact copy of the calling process (parent process). The child process inherits the following attributes from the parent process:

- Environment
- Close-on-exec flags (described in the exec ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutine)
- Signal handling settings (such as the SIG_DFL value, the SIG_IGN value, and the Function Address
 parameter)
- · Set user ID mode bit
- · Set group ID mode bit
- · Profiling on and off status
- · Nice value
- · All attached shared libraries
- · Process group ID
- tty group ID (described in the exit ("exit, atexit, _exit, or _Exit Subroutine" on page 203), atexit, or _exit subroutine, signal subroutine, and raise subroutine)
- Current directory
- Root directory

- File-mode creation mask (described in the **umask** subroutine)
- File size limit (described in the ulimit subroutine)
- Attached shared memory segments (described in the **shmat** subroutine)
- Attached mapped file segments (described in the shmat subroutine)
- Debugger process ID and multiprocess flag if the parent process has multiprocess debugging enabled (described in the ptrace ("ptrace, ptracex, ptrace64 Subroutine" on page 978) subroutine).

The child process differs from the parent process in the following ways:

- The child process has only one user thread; it is the one that called the fork subroutine.
- The child process has a unique process ID.
- · The child process ID does not match any active process group ID.
- · The child process has a different parent process ID.
- The child process has its own copy of the file descriptors for the parent process. However, each file descriptor of the child process shares a common file pointer with the corresponding file descriptor of the parent process.
- All **semadj** values are cleared. For information about **semadj** values, see the **semop** subroutine.
- · Process locks, text locks, and data locks are not inherited by the child process. For information about locks, see the **plock** ("plock Subroutine" on page 816) subroutine.
- If multiprocess debugging is turned on, the trace flags are inherited from the parent; otherwise, the trace flags are reset. For information about request 0, see the ptrace ("ptrace, ptracex, ptrace64 Subroutine" on page 978) subroutine.
- The child process utime, stime, cutime, and cstime subroutines are set to 0. (For more information, see the **getrusage** ("getrusage, getrusage64, times, or vtimes Subroutine" on page 360), **times**, and vtimes subroutines.)
- Any pending alarms are cleared in the child process. (For more information, see the incinterval) ("getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329), setitimer ("getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329), and alarm ("getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329) subroutines.)
- · The set of signals pending for the child process is initialized to the empty set.
- The child process can have its own copy of the message catalogue for the parent process.
- The set of signals pending for the child process is initialized as an empty set.

Attention: If you are using the fork or vfork subroutines with an Enhanced X-Windows, X Toolkit, or Motif application, open a separate display connection (socket) for the forked process. If the child process uses the same display connection as the parent, the X Server will not be able to interpret the resulting data.

The **f fork** subroutine is similar to **fork**, except for:

• It is required that the child process calls one of the exec functions immediately after it is created. Since the fork handlers are never called, the application data, mutexes and the locks are all undefined in the child process.

The vfork subroutine is supported as a compatibility interface for older Berkeley Software Distribution (BSD) system programs and can be used by compiling with the Berkeley Compatibility Library (libbsd.a).

In the Version 4 of the operating system, the parent process does not have to wait until the child either exits or executes, as it does in BSD systems. The child process is given a new address space, as in the fork subroutine. The child process does not share any parent address space.

Attention: When using the fork or vfork subroutines with an Enhanced X-Windows, X Toolkit, or Motif application, a separate display connection (socket) should be opened for the forked process. The child process should never use the same display connection as the parent. Display connections are embodied with sockets, and sockets are inherited by the child process. Any attempt to have multiple processes writing to the same display connection results in the random interleaving of X protocol packets at the word level. The resulting data written to the socket will not be valid or undefined X protocol packets, and the X Server will not be able to interpret it.

Attention: Although the fork and vfork subroutine may be used with Graphics Library applications, the child process must not make any additional Graphics Library subroutine calls. The child application inherits some, but not all of the graphics hardware resources of the parent. Drawing by the child process may hang the graphics adapter, the Enhanced X Server, or may cause unpredictable results and place the system into an unpredictable state.

For additional information, see the /usr/lpp/GL/README file.

Return Values

Upon successful completion, the **fork** subroutine returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the errno global variable is set to indicate the error.

Error Codes

The **fork** subroutine is unsuccessful if one or more of the following are true:

EAGAIN Exceeds the limit on the total number of processes running either systemwide or by a single user.

or the system does not have the resources necessary to create another process.

ENOMEM Not enough space exists for this process.

EPROCLIM If WLM is running, the limit on the number of processes or threads in the class may have been

met.

Related Information

The "getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329, "bindprocessor Subroutine" on page 99, "exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196, "exit, atexit, _exit, or _Exit Subroutine" on page 203, "getrusage, getrusage64, times, or vtimes Subroutine" on page 360, "getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329, "getpriority, setpriority, or nice Subroutine" on page 349, "plock Subroutine" on page 816, "pthread_atfork Subroutine" on page 905, "ptrace, ptracex, ptrace64 Subroutine" on page 978, raise subroutine, semop subroutine, "getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329, shmat subroutine, setpriority or getpriority ("getpriority, setpriority, or nice Subroutine" on page 349) subroutine, sigaction, sigvec, or signal subroutine, ulimit subroutine, umask subroutine, wait, waitpid, or wait3 subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Process Duplication and Termination in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging ProgramsLK provides more information about forking a multi-threaded process.

fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp disable Subroutine

Purpose

These subroutines allow operations on the floating-point trap control.

Library

Standard C Library (libc.a)

Syntax

```
#include <fptrap.h>
int fp_any_enable()
int fp is enabled( Mask)
fptrap t Mask;
void fp enable all()
void fp_enable(Mask)
fptrap_t Mask;
void fp disable all()
void fp disable(Mask)
fptrap t Mask;
```

Description

Floating point traps must be enabled before traps can be generated. These subroutines aid in manipulating floating-point traps and identifying the trap state and type.

In order to take traps on floating point exceptions, the fp trap subroutine must first be called to put the process in serialized state, and the fp_enable subroutine or fp_enable_all subroutine must be called to enable the appropriate traps.

The header file **fptrap.h** defines the following names for the individual bits in the floating-point trap control:

TRP_INVALID **Invalid Operation Summary**

TRP DIV BY ZERO Divide by Zero TRP_OVERFLOW Overflow Underflow TRP_UNDERFLOW TRP INEXACT Inexact Result

Parameters

Mask A 32-bit pattern that identifies floating-point traps.

Return Values

The **fp** any **enable** subroutine returns 1 if any floating-point traps are enabled. Otherwise, 0 is returned.

The fp_is_enabled subroutine returns 1 if the floating-point traps specified by the Mask parameter are enabled. Otherwise, 0 is returned.

The **fp enable all** subroutine enables all floating-point traps.

The **fp_enable** subroutine enables all floating-point traps specified by the *Mask* parameter.

The **fp disable all** subroutine disables all floating-point traps.

The **fp_disable** subroutine disables all floating-point traps specified by the *Mask* parameter.

Related Information

The fp_clr_flag, fp_set_flag, fp_read_flag, fp_swap_flag ("fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag Subroutine")subroutine, fp_invalid_op, fp_divbyzero, fp_overflow, fp_underflow, fp_inexact, fp_any_xcp ("fp_invalid_op, fp_divbyzero, fp_overflow, fp_underflow, fp_inexact, fp_any_xcp Subroutine" on page 256) subroutines, fp_iop_snan, fp_iop_infsinf, fp_iop_infdinf, fp_iop_zrdzr, fp_iop_infmzr, fp_iop_invcmp ("fp_iop_snan, fp_iop_infsinf, fp_iop_infdinf, fp_iop_zrdzr, fp_iop_infmzr, fp iop invcmp, fp iop sqrt, fp iop convert, or fp iop vxsoft Subroutines" on page 257) subroutines, fp_read_rnd, and fp_swap_rnd ("fp_read_rnd or fp_swap_rnd Subroutine" on page 259) subroutines, fp_trap ("fp_trap Subroutine" on page 263) subroutine.

Floating-Point Processor in Assembler Language Reference.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fp clr flag, fp set flag, fp read flag, or fp swap flag Subroutine

Purpose

Allows operations on the floating-point exception flags.

Library

Standard C Library (libc.a)

Syntax

```
#include <float.h>
#include <fpxcp.h>
void fp clr flag( Mask)
fpflag t Mask;
void fp set flag(Mask)
fpflag_t Mask;
fpflag_t fp_read_flag( )
fpflag_t fp_swap_flag(Mask)
fpflag_t Mask;
```

Description

These subroutines aid in determining both when an exception has occurred and the exception type. These subroutines can be called explicitly around blocks of code that may cause a floating-point exception.

According to the IEEE Standard for Binary Floating-Point Arithmetic, the following types of floating-point operations must be signaled when detected in a floating-point operation:

- · Invalid operation
- · Division by zero
- Overflow
- Underflow
- Inexact

An invalid operation occurs when the result cannot be represented (for example, a sqrt operation on a number less than 0).

The IEEE Standard for Binary Floating-Point Arithmetic states: "For each type of exception, the implementation shall provide a status flag that shall be set on any occurrence of the corresponding exception when no corresponding trap occurs. It shall be reset only at the user's request. The user shall be able to test and to alter the status flags individually, and should further be able to save and restore all five at one time."

Floating-point operations can set flags in the floating-point exception status but cannot clear them. Users can clear a flag in the floating-point exception status using an explicit software action such as the fp_swap_flag (0) subroutine.

The **fpxcp.h** file defines the following names for the flags indicating floating-point exception status:

FP INVALID Invalid operation summary

FP_OVERFLOW Overflow FP_UNDERFLOW Underflow FP_DIV_BY_ZERO Division by 0 FP_INEXACT Inexact result

In addition to these flags, the operating system supports additional information about the cause of an invalid operation exception. The following flags also indicate floating-point exception status and defined in the fpxcp.h file. The flag number for each exception type varies, but the mnemonics are the same for all ports. The following invalid operation detail flags are not required for conformance to the IEEE floating-point exceptions standard:

FP_INV_SNAN Signaling NaN FP_INV_ISI INF - INF FP INV IDI INF / INF FP_INV_ZDZ 0/0 FP INV IMZ INF x 0

FP_INV_CMP

Unordered compare Square root of a negative number FP_INV_SQRT FP_INV_CVI Conversion to integer error

FP_INV_VXSOFT Software request

Parameters

Mask A 32-bit pattern that identifies floating-point exception flags.

Return Values

The **fp_clr_flag** subroutine resets the exception status flags defined by the *Mask* parameter to 0 (false). The remaining flags in the exception status are unchanged.

The fp_set_flag subroutine sets the exception status flags defined by the Mask parameter to 1 (true). The remaining flags in the exception status are unchanged.

The fp_read_flag subroutine returns the current floating-point exception status. The flags in the returned exception status can be tested using the flag definitions above. You can test individual flags or sets of flags.

The **fp** swap flag subroutine writes the *Mask* parameter into the floating-point status and returns the floating-point exception status from before the write.

Users set or reset multiple exception flags using fp_set_flag and fp_clr_flag by ANDing or ORing definitions for individual flags. For example, the following resets both the overflow and inexact flags: fp clr flag (FP OVERFLOW | FP INEXACT)

Related Information

The fp any enable, fp is enabled, fp enable all, fp enable, fp disable, or fp disable all ("fp any enable, fp is enabled, fp enable all, fp enable, fp disable all, or fp disable Subroutine" on page 251) subroutine, fp any xcp, fp divbyzero, fp inexact, fp invalid op, fp overflow, fp underflow ("fp invalid op, fp divbyzero, fp overflow, fp underflow, fp inexact, fp any xcp Subroutine" on page 256) subroutines, fp iop infdinf, fp iop infmzr, fp iop infsinf, fp iop invcmp, fp iop snan, or fp iop zrdzr ("fp iop snan, fp iop infsinf, fp iop infdinf, fp iop zrdzr, fp iop infmzr, fp iop invcmp, fp iop sqrt, fp iop convert, or fp iop vxsoft Subroutines" on page 257) subroutines, fp read rnd or fp swap rnd ("fp read rnd or fp swap rnd Subroutine" on page 259) subroutine.

Floating-Point Exceptions Overview and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fp_cpusync Subroutine

Purpose

Queries or changes the floating-point exception enable (FE) bit in the Machine Status register (MSR).

Note: This subroutine has been replaced by the fp trapstate ("fp trapstate Subroutine" on page 265) subroutine. The fp cpusync subroutine is supported for compatibility, but the fp trapstate subroutine should be used for development.

Library

Standard C Library (libc.a)

Syntax

```
#include <fptrap.h>
int fp_cpusync ( Flag);
int Flag;
```

Description

The **fp_cpusync** subroutine is a service routine used to query, set, or reset the Machine Status Register (MSR) floating-point exception enable (FE) bit. The MSR FE bit determines whether a processor runs in pipeline or serial mode. Floating-point traps can only be generated by the hardware when the processor is in synchronous mode.

The **fp cpusync** subroutine changes only the MSR FE bit. It is a service routine for use in developing custom floating-point exception-handling software. If you are using the fp_enable or fp_enable_all ("fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine" on page 251) subroutine or the fp_sh_trap_info or fp_sh_set_stat ("fp_sh_info, fp_sh_trap_info, or fp sh set stat Subroutine" on page 260) subroutine, you must use the fp trap ("fp trap Subroutine" on page 263) subroutine to place the process in serial mode.

Parameters

Flag

Specifies to query or modify the MSR FE bit:

FP SYNC OFF

Sets the FE bit in the MSR to Off, which disables floating-point exception processing immediately.

FP SYNC ON

Sets the FE bit in the MSR to On, which enables floating-exception processing for the next floating-point operation.

FP SYNC QUERY

Returns the current state of the process (either FP_SYNC_ON or FP_SYNC_OFF) without modifying it.

If called with any other value, the fp cpusync subroutine returns FP SYNC ERROR.

Return Values

If called with the FP SYNC OFF or FP SYNC ON flag, the fp cpusync subroutine returns a value indicating which flag was in the previous state of the process.

If called with the FP_SYNC _QUERY flag, the fp_cpusync subroutine returns a value indicating the current state of the process, either the FP_SYNC_OFF or FP_SYNC_ON flag.

Error Codes

If the fp_cpusync subroutine is called with an invalid parameter, the subroutine returns **FP SYNC ERROR**. No other errors are reported.

Related Information

The fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable ("fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine" on page 251) subroutine, fp_clr_flag, fpset_flag, fp_read_flag, or fp_swap_flag ("fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag Subroutine" on page 252) subroutine, sigaction, sigvec, or signal subroutine.

Floating-Point Processor in Assembler Language Reference.

Floating-Point Exceptions in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fp flush_imprecise Subroutine

Purpose

Forces imprecise signal delivery.

Library

Standard C Library (libc.a)

Syntax

#include <fptrap.h> void fp_flush_imprecise ()

Description

The **fp_flush_imprecise** subroutine forces any imprecise interrupts to be reported. To ensure that no signals are lost when a program voluntarily exits, use this subroutine in combination with the **atexit** ("exit, atexit, _exit, or _Exit Subroutine" on page 203) subroutine.

Example

The following example illustrates using the **atexit** subroutine to run the **fp_flush_imprecise** subroutine before a program exits:

```
#include <fptrap.h>
#include <stdlib.h>
#include <stdio.h>
   if (0!=atexit(fp_flush_imprecise))
        puts ("Failure in atexit(fp_flush_imprecise) ");
```

Related Information

The atexit ("exit, atexit, _exit, or _Exit Subroutine" on page 203) subroutine, fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable ("fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine" on page 251) subroutine, fp_clr_flag, fp_read_flag, fp_swap_flag, or fpset_flag ("fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag Subroutine" on page 252) subroutine, fp_cpusync ("fp_cpusync Subroutine" on page 254) subroutine, fp_trap ("fp_trap Subroutine" on page 263) subroutine, sigaction subroutine.

Floating-Point Exceptions in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fp_invalid_op, fp_divbyzero, fp_overflow, fp_underflow, fp_inexact, fp_any_xcp Subroutine

Purpose

Tests to see if a floating-point exception has occurred.

Library

Standard C Library (libc.a)

Syntax

```
#include <float.h>
#include <fpxcp.h>
int
fp_invalid_op()
int fp_divbyzero()
int fp_overflow()
int fp_underflow()
int
fp_inexact()
int fp_any_xcp()
```

Description

These subroutines aid in determining when an exception has occurred and the exception type. These subroutines can be called explicitly after blocks of code that may cause a floating-point exception.

Return Values

The fp invalid_op subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_divbyzero** subroutine returns a value of 1 if a floating-point divide-by-zero exception status flag is set. Otherwise, a value of 0 is returned.

The fp_overflow subroutine returns a value of 1 if a floating-point overflow exception status flag is set. Otherwise, a value of 0 is returned.

The **fp underflow** subroutine returns a value of 1 if a floating-point underflow exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_inexact** subroutine returns a value of 1 if a floating-point inexact exception status flag is set. Otherwise, a value of 0 is returned.

The **fp any xcp** subroutine returns a value of 1 if a floating-point invalid operation, divide-by-zero, overflow, underflow, or inexact exception status flag is set. Otherwise, a value of 0 is returned.

Related Information

The fp any enable, fp is enabled, fp enable all, fp enable fp disable all, or fp disable ("fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable all, or fp_disable Subroutine" on page 251) subroutine, fp_clr_flag, fp_read_flag, fp_set_flag, or fp_swap_flag ("fp_clr_flag, fp_set_flag, fp read flag, or fp swap flag Subroutine" on page 252) subroutine, fp read rnd or fp swap rnd ("fp_read_rnd or fp_swap_rnd Subroutine" on page 259) subroutine.

Floating-Point Processor in Assembler Language Reference.

Floating-Point Exceptions and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fp_iop_snan, fp_iop_infsinf, fp_iop_infdinf, fp_iop_zrdzr, fp_iop_infmzr, fp_iop_invcmp, fp_iop_sqrt, fp_iop_convert, or fp_iop_vxsoft **Subroutines**

Purpose

Tests to see if a floating-point exception has occurred.

Library

Standard C Library (libc.a)

Syntax

```
#include <float.h>
#include <fpxcp.h>
int fp iop snan()
int fp_iop_infsinf()
fp iop infdinf()
int fp_iop_zrdzr()
fp iop infmzr()
int fp_iop_invcmp()
```

```
int
fp_iop_sqrt()
int fp_iop_convert()
int
fp iop vxsoft ();
```

Description

These subroutines aid in determining when an exception has occurred and the exception type. These subroutines can be called explicitly after blocks of code that may cause a floating-point exception.

Return Values

The **fp_iop_snan** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to a signaling NaN (NaNS) flag. Otherwise, a value of 0 is returned.

The **fp_iop_infsinf** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to an INF-INF flag. Otherwise, a value of 0 is returned.

The **fp_iop_infdinf** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to an INF/INF flag. Otherwise, a value of 0 is returned.

The **fp_iop_zrdzr** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to a 0.0/0.0 flag. Otherwise, a value of 0 is returned.

The **fp_iop_infmzr** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to an INF*0.0 flag. Otherwise, a value of 0 is returned.

The **fp_iop_invcmp** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to a compare involving a NaN. Otherwise, a value of 0 is returned.

The **fp_iop_sqrt** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to the calculation of a square root of a negative number. Otherwise, a value of 0 is returned.

The **fp_iop_convert** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to the conversion of a floating-point number to an integer, where the floating-point number was a NaN, an INF, or was outside the range of the integer. Otherwise, a value of 0 is returned.

The **fp_iop_vxsoft** subroutine returns a value of 1 if the VXSOFT detail bit is on. Otherwise, a value of 0 is returned.

fp_raise_xcp Subroutine

Purpose

Generates a floating-point exception.

Library

Standard C Library (libc.a)

Syntax

```
#include <fpxcp.h>
int fp_raise_xcp( mask)
fpflag_t mask;
```

Description

The fp_raise_xcp subroutine causes any floating-point exceptions defined by the mask parameter to be raised immediately. If the exceptions defined by the mask parameter are enabled and the program is running in serial mode, the signal for floating-point exceptions, SIGFPE, is raised.

If more than one exception is included in the mask variable, the exceptions are raised in the following order:

- 1. Invalid
- 2. Dividebyzero
- 3. Underflow
- 4. Overflow
- Inexact

Thus, if the user exception handler does not disable further exceptions, one call to the fp_raise_xcp subroutine can cause the exception handler to be entered many times.

Parameters

mask

Specifies a 32-bit pattern that identifies floating-point traps.

Return Values

The fp_raise_xcp subroutine returns 0 for normal completion and returns a nonzero value if an error occurs.

Related Information

The fp any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable ("fp any enable, fp is enabled, fp enable all, fp enable, fp disable all, or fp disable Subroutine" on page 251) subroutine, fp_clr_flag, fp_read_flag, fp_swap_flag, or fpset_flag ("fp_clr_flag, fp_set_flag, fp read flag, or fp swap flag Subroutine" on page 252) subroutine, fp cpusync ("fp cpusync Subroutine" on page 254) subroutine, **fp_trap** ("fp_trap Subroutine" on page 263) subroutine, **sigaction** subroutine.

fp read rnd or fp swap rnd Subroutine

Purpose

Read and set the IEEE floating-point rounding mode.

Library

Standard C Library (libc.a)

Syntax

```
#include <float.h>
```

```
fprnd_t fp_read_rnd()
fprnd t fp swap rnd( RoundMode)
fprnd t RoundMode;
```

Description

The fp read rnd subroutine returns the current rounding mode. The fp swap rnd subroutine changes the rounding mode to the RoundMode parameter and returns the value of the rounding mode before the change.

Floating-point rounding occurs when the infinitely precise result of a floating-point operation cannot be represented exactly in the destination floating-point format (such as double-precision format).

The IEEE Standard for Binary Floating-Point Arithmetic allows floating-point numbers to be rounded in four different ways: round toward zero, round to nearest, round toward +INF, and round toward -INF. Once a rounding mode is selected it affects all subsequent floating-point operations until another rounding mode is selected.

Note: The default floating-point rounding mode is round to nearest. All C main programs begin with the rounding mode set to round to nearest.

The encodings of the rounding modes are those defined in the ANSI C Standard. The **float.h** file contains definitions for the rounding modes. Below is the **float.h** definition, the ANSI C Standard value, and a description of each rounding mode.

float.h Definition	ANSI Value	Description
FP_RND_RZ	0	Round toward 0
FP_RND_RN	1	Round to nearest
FP_RND_RP	2	Round toward +INF
FP_RND_RM	3	Round toward -INF

The **fp_swap_rnd** subroutine can be used to swap rounding modes by saving the return value from **fp_swap_rnd**(*RoundMode*). This can be useful in functions that need to force a specific rounding mode for use during the function but wish to restore the caller's rounding mode on exit. Below is a code fragment that accomplishes this action:

```
save_mode = fp_swap_rnd (new_mode);
....desired code using new_mode
(void) fp swap rnd(save mode); /*restore caller's mode*/
```

Parameters

RoundMode

Specifies one of the following modes: FP_RND_RZ, FP_RND_RN, FP_RND_RP, or FP_RND_RM.

Related Information

The floor, ceil, nearest, trunc, rint, itrunc, uitrunc, fmod, or fabs ("floor, floorf, floorl, nearest, trunc, itrunc, or uitrunc Subroutine" on page 234) subroutine, fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable_all, or fp_disable ("fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine" on page 251) subroutine, fp_clr_flag, fp_read_flag, fp_set_flag, or fp_swap_flag ("fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag Subroutine" on page 252) subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fp_sh_info, fp_sh_trap_info, or fp_sh_set_stat Subroutine

Purpose

From within a floating-point signal handler, determines any floating-point exception that caused the trap in the process and changes the state of the Floating-Point Status and Control register (FPSCR) in the user process.

Library

Standard C Library (libc.a)

Syntax

```
#include <fpxcp.h>
#include <fptrap.h>
#include <signal.h>
void fp_sh_info( scp, fcp, struct size)
struct sigcontext *scp;
struct fp_sh_info *fcp;
size_t struct size;
void fp_sh_trap_info( scp, fcp)
struct sigcontext *scp;
struct fp_ctx *fcp;
void fp_sh_set_stat( scp, fpscr)
struct sigcontext *scp;
fpstat_t fpscr;
```

Description

These subroutines are for use within a user-written signal handler. They return information about the process that was running at the time the signal occurred, and they update the Floating-Point Status and Control register for the process.

Note: The fp_sh_trap_info subroutine is maintained for compatibility only. It has been replaced by the fp_sh_info subroutine, which should be used for development.

These subroutines operate only on the state of the user process that was running at the time the signal was delivered. They read and write the sigcontext structure. They do not change the state of the signal handler process itself.

The state of the signal handler process can be modified by the fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable subroutine.

fp sh info

The **fp** sh **info** subroutine returns information about the process that caused the trap by means of a floating-point context (fp_sh_info) structure. This structure contains the following information:

```
typedef struct fp sh info {
fpstat t
               fpscr;
fpflag_t
               trap;
short
               trap_mode;
char
               flags;
char
               extra;
} fp sh info t;
```

The fields are:

trap

fpscr The Floating-Point Status and Control register (FPSCR) in the user process at the time the

interrupt occurred.

A mask indicating the trap or traps that caused the signal handler to be entered. This mask is the logical OR operator of the enabled floating-point exceptions that occurred to cause the trap. This mask can have up to two exceptions; if there are two, the INEXACT signal must be one of them. If the mask is 0, the SIGFPE signal was raised not by a floating-point operation, but by the kill or raise subroutine or the kill command.

trap mode

flags

The trap mode in effect in the process at the time the signal handler was entered. The values returned in the **fp_sh_info.trap_mode** file use the following argument definitions:

FP_TRAP_OFF

Trapping off

FP_TRAP_SYNC

Precise trapping on

FP TRAP IMP REC

Recoverable imprecise trapping on

FP TRAP IMP

Non-recoverable imprecise trapping on

This field is interpreted as an array of bits and should be accessed with masks. The following

mask is defined:

FP IAR STAT

If the value of the bit at this mask is 1, the exception was precise and the IAR points to the instruction that caused the exception. If the value bit at this mask is 0, the exception

was imprecise.

fp_sh_trap_info

The fp_sh_trap_info subroutine is maintained for compatibility only. The fp_sh_trap_info subroutine returns information about the process that caused the trap by means of a floating-point context (fp_ctx) structure. This structure contains the following information:

fpstat t fpscr; fpflag t trap;

The fields are:

The Floating-Point Status and Control register (FPSCR) in the user process at the time the fpscr

interrupt occurred.

A mask indicating the trap or traps that caused the signal handler to be entered. This mask is the trap

logical OR operator of the enabled floating-point exceptions that occurred to cause the trap. This mask can have up to two exceptions; if there are two, the INEXACT signal must be one of them. If the mask is 0, the SIGFPE signal was raised not by a floating-point operation, but by the kill or

raise subroutine or the kill command.

fp sh set stat

The fp_sh_set_stat subroutine updates the Floating-Point Status and Control register (FPSCR) in the user process with the value in the fpscr field.

The signal handler must either clear the exception bit that caused the trap to occur or disable the trap to prevent a recurrence. If the instruction generated more than one exception, and the signal handler clears only one of these exceptions, a signal is raised for the remaining exception when the next floating-point instruction is executed in the user process.

Parameters

Specifies a floating-point context structure. fcp Specifies a **sigcontext** structure for the interrupt. scp Specifies the size of the fp_sh_info structure. struct_size

Specifies which Floating-Point Status and Control register to update. fpscr

Related Information

The fp any_enable, fp_disable_all, fp_disable, fp_enable_all, fp_enable, or fp_is_enabled ("fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine" on page 251) subroutine, fp_clr_flag, fp_read_flag, fp_set_flag, or fp_swap_flag ("fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag Subroutine" on page 252) subroutine, fp_trap ("fp_trap Subroutine") subroutine.

Floating-Point Exceptions in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fp trap Subroutine

Purpose

Queries or changes the mode of the user process to allow floating-point exceptions to generate traps.

Library

Standard C Library (libc.a)

Syntax

#include <fptrap.h>

int fp trap(flag) int flag;

Description

The fp_trap subroutine queries and changes the mode of the user process to allow or disallow floating-point exception trapping. Floating-point traps can only be generated when a process is executing in a traps-enabled mode.

The default state is to execute in pipelined mode and not to generate floating-point traps.

Note: The fp trap routines only change the execution state of the process. To generate floating-point traps, you must also enable traps. Use the fp_enable ("fp_any_enable, fp_is_enabled, fp enable all, fp enable, fp disable all, or fp disable Subroutine" on page 251) and fp enable all subroutines to enable traps.

Before calling the fp trap(FP TRAP SYNC) routine, previous floating-point operations can set to True certain exception bits in the Floating-Point Status and Control register (FPSCR). Enabling these Cexceptions and calling the fp trap(FP TRAP SYNC) routine does not cause an immediate trap to occur. That is, the operation of these traps is edge-sensitive, not level-sensitive.

The fp trap subroutine does not clear the exception history. You can guery this history by using any of the following subroutines:

- fp any xcp
- fp divbyzero
- fp_iop_convert
- fp_iop_infdinf
- fp_iop_infmzr
- fp iop infsinf
- fp_iop_invcmp
- fp_iop_snan

- fp_iop_sqrt
- · fp iop vxsoft
- fp_iop_zrdzr
- fp_inexact
- fp invalid op
- fp_overflow
- · fp_underflow

Parameters

flag Specifies a query of or change in the mode of the user process:

FP TRAP OFF

Puts the user process into trapping-off mode and returns the previous mode of the process, either FP_TRAP_SYNC, FP_TRAP_IMP, FP_TRAP_IMP_REC, or FP_TRAP_OFF.

FP_TRAP_QUERY

Returns the current mode of the user process.

FP TRAP SYNC

Puts the user process into precise trapping mode and returns the previous mode of the process.

FP TRAP IMP

Puts the user process into non-recoverable imprecise trapping mode and returns the previous mode.

FP_TRAP_IMP_REC

Puts the user process into recoverable imprecise trapping mode and returns the previous

FP_TRAP_FASTMODE

Puts the user process into the fastest trapping mode available on the hardware platform.

Note: Some hardware models do not support all modes. If an unsupported mode is requested, the fp_trap subroutine returns FP_TRAP_UNIMPL.

Return Values

If called with the FP_TRAP_OFF, FP_TRAP_IMP, FP_TRAP_IMP_REC, or FP_TRAP_SYNC flag, the fp trap subroutine returns a value indicating which flag was in the previous mode of the process if the hardware supports the requested mode. If the hardware does not support the requested mode, the fp trap subroutine returns FP TRAP UNIMPL.

If called with the FP TRAP QUERY flag, the fp trap subroutine returns a value indicating the current mode of the process, either the FP_TRAP_OFF, FP_TRAP_IMP, FP_TRAP_IMP_REC, or FP_TRAP_SYNC flag.

If called with FP TRAP FASTMODE, the fp trap subroutine sets the fastest mode available and returns the mode selected.

Error Codes

If the fp trap subroutine is called with an invalid parameter, the subroutine returns FP TRAP ERROR.

If the requested mode is not supported on the hardware platform, the subroutine returns FP_TRAP_UNIMPL.

fp_trapstate Subroutine

Purpose

Queries or changes the trapping mode in the Machine Status register (MSR).

Note: This subroutine replaces the fp_cpusync ("fp_cpusync Subroutine" on page 254) subroutine. The fp cpusync subroutine is supported for compatibility, but the fp trapstate subroutine should be used for development.

Library

Standard C Library (libc.a)

Syntax

#include <fptrap.h> int fp_trapstate (int)

Description

The **fp_trapstate** subroutine is a service routine used to query or set the trapping mode. The trapping mode determines whether floating-point exceptions can generate traps, and can affect execution speed. See Floating-Point Exceptions Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs for a description of precise and imprecise trapping modes. Floating-point traps can be generated by the hardware only when the processor is in a traps-enabled mode.

The fp_trapstate subroutine changes only the trapping mode. It is a service routine for use in developing custom floating-point exception-handling software. If you are using the fp enable ("fp any enable, fp is enabled, fp enable all, fp enable, fp disable all, or fp disable Subroutine" on page 251) or fp_enable_all subroutine or the fp_sh_info ("fp_sh_info, fp_sh_trap_info, or fp_sh_set_stat Subroutine" on page 260) or **fp sh set stat** subroutine, you must use the **fp trap** ("fp trap Subroutine" on page 263) subroutine to change the process' trapping mode.

Parameters

flag Specifies a query of, or change in, the trap mode:

FP TRAPSTATE OFF

Sets the trapping mode to Off and returns the previous mode.

FP_TRAPSTATE_QUERY

Returns the current trapping mode without modifying it.

FP TRAPSTATE IMP

Puts the process in non-recoverable imprecise trapping mode and returns the previous

FP_TRAPSTATE_IMP_REC

Puts the process in recoverable imprecise trapping mode and returns the previous state.

FP TRAPSTATE PRECISE

Puts the process in precise trapping mode and returns the previous state.

FP_TRAPSTATE_FASTMODE

Puts the process in the fastest trap-generating mode available on the hardware platform and returns the state selected.

Note: Some hardware models do not support all modes. If an unsupported mode is requested, the fp trapstate subroutine returns FP TRAP UNIMPL and the trapping mode is not changed.

Return Values

If called with the FP_TRAPSTATE_OFF, FP_TRAPSTATE_IMP, FP_TRAPSTATE_IMP_REC, or FP_TRAPSTATE_PRECISE flag, the fp_trapstate subroutine returns a value indicating the previous mode of the process. The value may be FP_TRAPSTATE_OFF, FP_TRAPSTATE_IMP, FP_TRAPSTATE_IMP_REC, or FP_TRAPSTATE_PRECISE. If the hardware does not support the requested mode, the fp_trapstate subroutine returns FP_TRAP_UNIMPL.

If called with the FP_TRAP_QUERY flag, the fp_trapstate subroutine returns a value indicating the current mode of the process. The value may be FP_TRAPSTATE_OFF, FP_TRAPSTATE_IMP, FP_TRAPSTATE_IMP_REC, or FP_TRAPSTATE_PRECISE.

If called with the FP_TRAPSTATE_FASTMODE flag, the fp_trapstate subroutine returns a value indicating which mode was selected. The value may be FP_TRAPSTATE_OFF, FP_TRAPSTATE_IMP, FP_TRAPSTATE_IMP_REC, or FP_TRAPSTATE_PRECISE.

Related Information

The fp_any_enable, fp_disable_all, fp_disable, fp_enable_all, fp_enable, or fp_is_enabled ("fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine" on page 251) subroutine, fp_clr_flag, fp_read_flag, fpset_flag, or fp_swap_flag ("fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag Subroutine" on page 252) subroutine, sigaction, signal, or sigvec subroutine.

The Floating-Point Processor in Assembler Language Reference.

Floating-Point Exceptions in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fpclassify Macro

Purpose

Classifies real floating type.

Syntax

#include <math.h>

int fpclassify(x)
real-floating x;

Description

The **fpclassify** macro classifies the *x* parameter as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. An argument represented in a format wider than its semantic type is converted to its semantic type. Classification is based on the type of the argument.

Parameters

x Specifies the value to be classified.

Return Values

The **fpclassify** macro returns the value of the number classification macro appropriate to the value of its argument.

Related Information

"isfinite Macro" on page 464, "isinf Subroutine" on page 466, "class, _class, finite, isnan, or unordered Subroutines" on page 138, "isnormal Macro" on page 468.

The signbit Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

math.h in AIX 5L Version 5.2 Files Reference.

fread or fwrite Subroutine

Purpose

Reads and writes binary files.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdio.h>
size_t fread ( (void *) Pointer, Size, NumberOfItems, Stream ("Parameters" on page 268))
size t Size, NumberOfItems ("Parameters" on page 268);
FILE *Stream ("Parameters" on page 268);
size t fwrite (Pointer, Size, NumberOfItems, Stream ("Parameters" on page 268))
const void *Pointer ("Parameters" on page 268);
size t Size, NumberOfItems ("Parameters" on page 268);
FILE *Stream ("Parameters" on page 268);
```

Description

The **fread** subroutine copies the number of data items specified by the *NumberOfItems* parameter from the input stream into an array beginning at the location pointed to by the *Pointer* parameter. Each data item has the form *Pointer.

The fread subroutine stops copying bytes if an end-of-file (EOF) or error condition is encountered while reading from the input specified by the Stream parameter, or when the number of data items specified by the NumberOfItems parameter have been copied. This subroutine leaves the file pointer of the Stream parameter, if defined, pointing to the byte following the last byte read. The fread subroutine does not change the contents of the Stream parameter.

The st atime field will be marked for update by the first successful run of the fgetc ("getc, getchar, fgetc, or getw Subroutine" on page 300), fgets ("gets or fgets Subroutine" on page 366), fgetwc, ("getwc, fgetwc, or getwchar Subroutine" on page 397), fgetws ("getws or fgetws Subroutine" on page 399), fread, fscanf, getc ("getc, getchar, fgetc, or getw Subroutine" on page 300), getchar ("getc, getchar, fgetc, or getw Subroutine" on page 300), gets ("gets or fgets Subroutine" on page 366), or scanf subroutine using a stream that returns data not supplied by a prior call to the ungetc or ungetwc subroutine.

Note: The fread subroutine is a buffered read subroutine library call. It reads data in 4KB blocks. For tape block sizes greater than 4KB, use the open ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine and read subroutine.

The **fwrite** subroutine writes items from the array pointed to by the *Pointer* parameter to the stream pointed to by the Stream parameter. Each item's size is specified by the Size parameter. The fwrite subroutine writes the number of items specified by the *NumberOfltems* parameter. The file-position

indicator for the stream is advanced by the number of bytes successfully written. If an error occurs, the resulting value of the file-position indicator for the stream is indeterminate.

The **fwrite** subroutine appends items to the output stream from the array pointed to by the *Pointer* parameter. The **fwrite** subroutine appends as many items as specified in the *NumberOfltems* parameter.

The fwrite subroutine stops writing bytes if an error condition is encountered on the stream, or when the number of items of data specified by the *NumberOfltems* parameter have been written. The **fwrite** subroutine does not change the contents of the array pointed to by the Pointer parameter.

The st ctime and st mtime fields will be marked for update between the successful run of the fwrite subroutine and the next completion of a call to the **fflush** ("fclose or fflush Subroutine" on page 213) or fclose subroutine on the same stream, the next call to the exit ("exit, atexit, _exit, or _Exit Subroutine" on page 203) subroutine, or the next call to the abort ("abort Subroutine" on page 3) subroutine.

Parameters

Pointer Points to an array.

Size Specifies the size of the variable type of the array pointed to by the *Pointer* parameter. The

Size parameter can be considered the same as a call to **sizeof** subroutine.

NumberOfItems Specifies the number of items of data. Stream Specifies the input or output stream.

Return Values

The **fread** and **fwrite** subroutines return the number of items actually transferred. If the *NumberOfItems* parameter contains a 0, no characters are transferred, and a value of 0 is returned. If the NumberOfItems parameter contains a negative number, it is translated to a positive number, since the NumberOfItems parameter is of the unsigned type.

Error Codes

If the fread subroutine is unsuccessful because the I/O stream is unbuffered or data needs to be read into the I/O stream's buffer, it returns one or more of the following error codes:

EAGAIN Indicates that the O_NONBLOCK flag is set for the file descriptor specified by the Stream

parameter, and the process would be delayed in the fread operation.

EBADF Indicates that the file descriptor specified by the Stream parameter is not a valid file descriptor

open for reading.

EINTR Indicates that the read operation was terminated due to receipt of a signal, and no data was

transferred.

Note: Depending upon which library routine the application binds to, this subroutine may return EINTR. Refer to the **signal** subroutine regarding **sa restart**.

EIO Indicates that the process is a member of a background process group attempting to

perform a read from its controlling terminal, and either the process is ignoring or blocking the

SIGTTIN signal or the process group has no parent process.

ENOMEM Indicates that insufficient storage space is available.

ENXIO Indicates that a request was made of a nonexistent device.

If the fwrite subroutine is unsuccessful because the I/O stream is unbuffered or the I/O stream's buffer needs to be flushed, it returns one or more of the following error codes:

EAGAIN Indicates that the O NONBLOCK flag is set for the file descriptor specified by the Stream

parameter, and the process is delayed in the write operation.

EBADF Indicates that the file descriptor specified by the Stream parameter is not a valid file

descriptor open for writing.

Indicates that an attempt was made to write a file that exceeds the file size of the process **EFBIG**

limit or the systemwide maximum file size.

Indicates that the write operation was terminated due to the receipt of a signal, and no data **EINTR**

was transferred.

EIO Indicates that the process is a member of a background process group attempting to perform

a write to its controlling terminal, the TOSTOP signal is set, the process is neither ignoring

nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.

ENOSPC Indicates that there was no free space remaining on the device containing the file.

EPIPE Indicates that an attempt is made to write to a pipe or first-in-first-out (FIFO) process that is

not open for reading by any process. A SIGPIPE signal is sent to the process.

The **fwrite** subroutine is also unsuccessful due to the following error conditions:

ENOMEM Indicates that insufficient storage space is available.

ENXIO Indicates that a request was made of a nonexistent device, or the request was outside the

capabilities of the device.

Related Information

The abort ("abort Subroutine" on page 3) subroutine, exit ("exit, atexit, exit, or Exit Subroutine" on page 203) subroutine, fflush or fclose ("fclose or fflush Subroutine" on page 213) subroutine, fopen, freopen, or **fdopen** ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, **getc**, getchar, fgetc, or getw ("getc, getchar, fgetc, or getw Subroutine" on page 300) subroutine, getwc, factwc, or getwchar ("getwc, factwc, or getwchar Subroutine" on page 397) subroutine, gets or facts ("gets or fgets Subroutine" on page 366) subroutine, getws or fgetws ("getws or fgetws Subroutine" on page 399) subroutine, open ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine, print, fprintf, or sprintf ("printf, fprintf, sprintf, sprintf, wsprintf, vfprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine, putc, putchar, fputc, or putw ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine, putwc, putwchar, or fputwc ("putwc, putwchar, or fputwc Subroutine" on page 995) subroutine, puts or fputs ("puts or fputs Subroutine" on page 994) subroutine. putws or fputws ("putws or fputws Subroutine" on page 997) subroutine, read subroutine, scanf, fscanf, sscanf, or wsscanf subroutine, ungetc or ungetwc subroutine, write subroutine.

The Input and Output Handling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

freehostent Subroutine

Purpose

To free memory allocated by getipnodebyname and getipnodebyaddr.

Library

Standard C Library (libc.a)

Syntax

#include <netdb.h> void freehostent (ptr)struct hostent * ptr;

Description

The **freehostent** subroutine frees any dynamic storage pointed to by elements of ptr. This includes the hostent structure and the data areas pointed to by the h name, h addr list, and h aliases members of the hostent structure.

Related Information

The **getipnodebyaddr** subroutine and **getipnodebyname** subroutine.

frevoke Subroutine

Purpose

Revokes access to a file by other processes.

Library

Standard C Library (libc.a)

Syntax

int frevoke (FileDescriptor) int FileDescriptor;

Description

The **frevoke** subroutine revokes access to a file by other processes.

All accesses to the file are revoked, except through the file descriptor specified by the FileDescriptor parameter to the frevoke subroutine. Subsequent attempts to access the file, using another file descriptor established before the frevoke subroutine was called, fail and cause the process to receive a return value of -1, and the errno global variable is set to EBADF.

A process can revoke access to a file only if its effective user ID is the same as the file owner ID or if the invoker has root user authority.

Note: The frevoke subroutine has no affect on subsequent attempts to open the file. To ensure exclusive access to the file, the caller should change the mode of the file before issuing the frevoke subroutine. Currently the frevoke subroutine works only on terminal devices.

Parameters

FileDescriptor A file descriptor returned by a successful **open** subroutine.

Return Values

Upon successful completion, the **frevoke** subroutine returns a value of 0.

If the frevoke subroutine fails, it returns a value of -1 and the errno global variable is set to indicate the error.

Error Codes

The **frevoke** subroutine fails if the following is true:

EBADF The *FileDescriptor* value is not the valid file descriptor of a terminal.

EPERM The effective user ID of the calling process is not the same as the file owner ID.

frexpf, frexpl, or frexp Subroutine

Purpose

Extracts the mantissa and exponent from a double precision number.

Syntax

```
#include <math.h>
float frexpf (num, exp)
float num;
int *exp;
long double frexpl (num, exp)
long double num;
int exp;
double frexp (num, exp)
double num;
int * exp;
```

Description

The frexpf, frexpl, and frexp subroutines break a floating-point number num into a normalized fraction and an integral power of 2. The integer exponent is stored in the **int** object pointed to by exp.

Parameters

Specifies the floating-point number to be broken into a normalized fraction and an integral power of 2. num Points to where the integer exponent is stored. exp

Return Values

For finite arguments, the **frexpf**, **frexpl**, and **frexp** subroutines return the value x, such that x has a magnitude in the interval $[\frac{1}{2}, 1)$ or 0, and *num* equals x times 2 raised to the power exp.

If *num* is NaN, a NaN is returned, and the value of *exp is unspecified.

If num is ± 0 , ± 0 is returned, and the value of *exp is 0.

If *num* is ±Inf, *num* is returned, and the value of *exp is unspecified.

Related Information

"class, _class, finite, isnan, or unordered Subroutines" on page 138 and "modf, modff, or modfl Subroutine" on page 695

math.h in AIX 5L Version 5.2 Files Reference.

fscntl Subroutine

Purpose

Controls file system control operations.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/types.h>
int fscntl ( vfs_id, Command, Argument,
                                            ArgumentSize)
int vfs_id;
int Command;
char *Argument;
int ArgumentSize;
```

Description

The fscntl subroutine performs a variety of file system-specific functions. These functions typically require root user authority.

At present, only one file system, the Journaled File System, supports any commands via the fscntl subroutine.

Note: Application programs should not call this function, which is reserved for system management commands such as the chfs command.

Parameters

vfs_id Identifies the file system to be acted upon. This information is returned by the stat

subroutine in the st vfs field of the stat.h file.

Command Identifies the operation to be performed.

Argument Specifies a pointer to a block of file system specific information that defines how the

operation is to be performed.

ArgumentSize Defines the size of the buffer pointed to by the *Argument* parameter.

Return Values

Upon successful completion, the fscntl subroutine returns a value of 0. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **fscntl** subroutine fails if one or both of the following are true:

EINVAL The vfs_id parameter does not identify a valid file system. **EINVAL** The Command parameter is not recognized by the file system.

Related Information

The chfs command.

The stat.h file.

Understanding File-System Helpers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs explains file system helpers and examines file system-helper execution syntax.

fseek, fseeko, fseeko64, rewind, ftell, ftello, ftello64, fgetpos, fgetpos64, fsetpos, or fsetpos64 Subroutine

Purpose

Repositions the file pointer of a stream.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdio.h>
int fseek ( Stream, Offset, Whence)
FILE *Stream;
long int Offset;
int Whence;
void rewind (Stream)
FILE *Stream;
long int ftell (Stream)
FILE *Stream;
int fgetpos (Stream, Position)
FILE *Stream;
fpos_t *Position;
int fsetpos (Stream, Position)
FILE *Stream;
const fpos_t *Position;
int fseeko ( Stream, Offset, Whence)
FILE *Stream;
off t Offset;
int Whence;
int fseeko64 ( Stream, Offset, Whence)
FILE *Stream;
off64_t Offset;
int Whence;
off t int ftello (Stream)
FILE *Stream;
off64 t int ftello64 (Stream)
FILE *Stream;
int fgetpos64 (Stream, Position)
FILE *Stream;
fpos64 t *Position;
int fsetpos64 (Stream, Position)
FILE *Stream;
const fpos64_t *Position;
```

Description

The fseek, fseeko and fseeko64 subroutines set the position of the next input or output operation on the I/O stream specified by the Stream parameter. The position if the next operation is determined by the Offset parameter, which can be either positive or negative.

The **fseek**, **fseeko** and **fseeko64** subroutines set the file pointer associated with the specified *Stream* as follows:

- If the Whence parameter is set to the SEEK_SET value, the pointer is set to the value of the Offset parameter.
- If the Whence parameter is set to the SEEK CUR value, the pointer is set to its current location plus the value of the *Offset* parameter.
- If the Whence parameter is set to the SEEK END value, the pointer is set to the size of the file plus the value of the *Offset* parameter.

The fseek, fseeko, and fseeko64 subroutine are unsuccessful if attempted on a file that has not been opened using the fopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine. In particular, the fseek subroutine cannot be used on a terminal or on a file opened with the popen ("popen Subroutine" on page 864) subroutine. The fseek and fseeko subroutines will also fail when the resulting offset is larger than can be properly returned.

The **rewind** subroutine is equivalent to calling the **fseek** subroutine using parameter values of (Stream, SEEK SET, SEEK SET), except that the rewind subroutine does not return a value.

The fseek, fseeko, fseeko64 and rewind subroutines undo any effects of the ungetc and ungetwo subroutines and clear the end-of-file (EOF) indicator on the same stream.

The fseek, fseeko, and fseeko64 function allows the file-position indicator to be set beyond the end of existing data in the file. If data is written later at this point, subsequent reads of data in the gap will return bytes of the value 0 until data is actually written into the gap.

A successful calls to the fsetpos or fsetpos64 subroutines clear the EOF indicator and undoes any effects of the **ungetc** and **ungetwc** subroutines.

After an fseek, fseeko, fseeko64 or a rewind subroutine, the next operation on a file opened for update can be either input or output.

ftell, ftello and ftello64 subroutines return the position current value of the file-position indicator for the stream pointed to by the Stream parameter. ftell and ftello will fail if the resulting offset is larger than can be properly returned.

The faetpos and faetpos64 subroutines store the current value of the file-position indicator for the stream pointed to by the Stream parameter in the object pointed to by the Position parameter. The fsetpos and fsetpos64 set the file-position indicator for Stream according to the value of the Position parameter, which must be the result of a prior call to fgetpos or fgetpos64 subroutine. fgetpos and fsetpos will fail if the resulting offset is larger than can be properly returned.

Parameters

Stream Specifies the input/output (I/O) stream.

Offset Determines the position of the next operation.

Whence Determines the value for the file pointer associated with the *Stream* parameter.

Position Specifies the value of the file-position indicator.

Return Values

Upon successful completion, the fseek, fseeko and fseeko64 subroutine return a value of 0. Otherwise, it returns a value of -1.

Upon successful completion, the ftell, ftello and ftello64 subroutine return the offset of the current byte relative to the beginning of the file associated with the named stream. Otherwise, a long int value of -1 is returned and the errno global variable is set.

Upon successful completion, the fgetpos, fgetpos64, fsetpos and fsetpos64 subroutines return a value of 0. Otherwise, a nonzero value is returned and the **errno** global variable is set to the specific error.

The **errno** global variable is used to determine if an error occurred during a **rewind** subroutine call.

Error Codes

If the fseek, fseeko, fseeko64, ftell, ftello, ftello64 or rewind subroutine are unsuccessful because the stream is unbuffered or the stream buffer needs to be flushed and the call to the subroutine causes an underlying **Iseek** or **write** subroutine to be invoked, it returns one or more of the following error codes:

EAGAIN Indicates that the O_NONBLOCK flag is set for the file descriptor, delaying the process in the

write operation.

EBADF Indicates that the file descriptor underlying the Stream parameter is not open for writing.

Indicates that an attempt has been made to write to a file that exceeds the file-size limit of the **EFBIG**

process or the maximum file size.

EFBIG Indicates that the file is a regular file and that an attempt was made to write at or beyond the

offset maximum associated with the corresponding stream.

EINTR Indicates that the write operation has been terminated because the process has received a

signal, and either no data was transferred, or the implementation does not report partial

transfers for this file.

EIO Indicates that the process is a member of a background process group attempting to perform

> a write subroutine to its controlling terminal, the TOSTOP flag is set, the process is not ignoring or blocking the SIGTTOU signal, and the process group of the process is orphaned.

This error may also be returned under implementation-dependent conditions.

ENOSPC Indicates that no remaining free space exists on the device containing the file.

EPIPE Indicates that an attempt has been made to write to a pipe or FIFO that is not open for

reading by any process. A SIGPIPE signal will also be sent to the process.

EINVAL Indicates that the Whence parameter is not valid. The resulting file-position indicator will be set

to a negative value. The EINVAL error code does not apply to the ftell and rewind

subroutines.

ESPIPE Indicates that the file descriptor underlying the Stream parameter is associated with a pipe or

FIFO.

EOVERFLOW Indicates that for fseek, the resulting file offset would be a value that cannot be represented

correctly in an object of type long.

EOVERFLOW Indicates that for fseeko, the resulting file offset would be a value that cannot be represented

correctly in an object of type *off_t*.

ENXIO Indicates that a request was made of a non-existent device, or the request was outside the

capabilities of the device.

The **fgetpos** and **fsetpos** subroutines are unsuccessful due to the following conditions:

EINVAL Indicates that either the Stream or the Position parameter is not valid. The EINVAL error code

does not apply to the fgetpos subroutine.

EBADF Indicates that the file descriptor underlying the Stream parameter is not open for writing. **ESPIPE** Indicates that the file descriptor underlying the Stream parameter is associated with a pipe or

FIFO.

The fseek, fseeko, ftell, ftello, fgetpos, and fsetpos subroutines are unsuccessful under the following condition:

EOVERFLOW The resulting could not be returned properly.

Related Information

The **closedir** ("opendir, readdir, telldir, seekdir, rewinddir, or closedir Subroutine" on page 771) subroutine, **fopen**, **fopen64**, **freopen**, **freopen64** or **fdopen** ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, **Iseek** or **Iseek64** ("Iseek, Ilseek or Iseek64 Subroutine" on page 646) subroutine, **opendir**, **readdir**, **rewinddir**, **seekdir**, or **telldir** ("opendir, readdir, telldir, seekdir, rewinddir, or closedir Subroutine" on page 771) subroutine, **popen** ("popen Subroutine" on page 864) subroutine, **ungetc** or **ungetwc** subroutine, **writex**, **writex**, or **writevx** subroutine.

Input and Output Handling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fsync or fsync_range Subroutine

Purpose

Writes changes in a file to permanent storage.

Library

Standard C Library (libc.a)

Syntax

```
#include <unistd.h>
int fsync ( FileDescriptor)
int FileDescriptor;
int fsync_range (FileDescriptor, how, start, length)
int FileDescriptor;
int how;
off_t start;
off t length;
```

Description

The **fsync** subroutine causes all modified data in the open file specified by the *FileDescriptor* parameter to be saved to permanent storage. On return from the **fsync** subroutine, all updates have been saved on permanent storage.

The **fsync_range** subroutine causes all modified data in the specified range of the open file specified by the *FileDescriptor* parameter to be saved to permanent storage. On return from the **fsync_range** subroutine, all updates in the specified range have been saved on permanent storage.

Data written to a file that a process has opened for deferred update (with the **O_DEFER** flag) is not written to permanent storage until another process issues an **fsync_range** or **fsync** call against this file or runs a synchronous **write** subroutine (with the **O_SYNC** flag) on this file. See the **fcntl.h** file and the **open** subroutine for descriptions of the **O_DEFER** and **O_SYNC** flags respectively.

Note: The file identified by the *FileDescriptor* parameter must be open for writing when the **fsync_range** or **fsync** subroutine is issued or the call is unsuccessful. This restriction was not enforced in BSD systems.

Parameters

FileDescriptor A valid, open file descriptor.

how How to flush, FDATASYNC, or FFILESYNC.

FDATASYNC

Write file data and enough of the meta-data to retrieve the data for the specified

range.

FFILESYNC

All modified file data and meta-data for the specified range.

Starting file offset. start

Length, or zero for everything. length

Return Values

Upon successful completion, the fsync subroutine returns a value of 0. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Upon successful completion, the **fsync range** subroutine returns a value of 0. Otherwise a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **fsync** subroutine is unsuccessful if one or more of the following are true:

EIO An I/O error occurred while reading from or writing to the file system. **EBADF** The FileDescriptor parameter is not a valid file descriptor open for writing.

EINVAL The file is not a regular file.

EINTR The **fsync** subroutine was interrupted by a signal.

Related Information

The open, openx, or creat ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine, sync subroutine, write, writex, writev, or writevx subroutine.

The fcntl.h file.

Files, Directories, and File Systems Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs contains information about i-nodes, file descriptors, file-space allocation, and more.

ftok Subroutine

Purpose

Generates a standard interprocess communication key.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok ( Path, ID)
```

char *Path; int ID;

Description

Attention: If the *Path* parameter of the **ftok** subroutine names a file that has been removed while keys still refer to it, the **ftok** subroutine returns an error. If that file is then re-created, the **ftok** subroutine will probably return a key different from the original one.

Attention: Each installation should define standards for forming keys. If standards are not adhered to, unrelated processes may interfere with each other's operation.

Attention: The **ftok** subroutine does not guarantee unique key generation. However, the occurrence of key duplication is very rare and mostly for across file systems.

The **ftok** subroutine returns a key, based on the *Path* and *ID* parameters, to be used to obtain interprocess communication identifiers. The **ftok** subroutine returns the same key for linked files if called with the same *ID* parameter. Different keys are returned for the same file if different *ID* parameters are used.

All interprocess communication facilities require you to supply a key to the **msgget**, **semget**, and **shmget** subroutines in order to obtain interprocess communication identifiers. The **ftok** subroutine provides one method for creating keys, but other methods are possible. For example, you can use the project ID as the most significant byte of the key, and use the remaining portion as a sequence number.

Parameters

Path Specifies the path name of an existing file that is accessible to the process.

ID Specifies a character that uniquely identifies a project.

Return Values

When successful, the **ftok** subroutine returns a key that can be passed to the **msgget**, **semget**, or **shmget** subroutine.

Error Codes

The ftok subroutine returns the value (key_t)-1 if one or more of the following are true:

- The file named by the Path parameter does not exist.
- The file named by the *Path* parameter is not accessible to the process.
- The *ID* parameter has a value of 0.

Related Information

The msgget ("msgget Subroutine" on page 714) subroutine, semget subroutine, shmget subroutine.

Subroutines Overview and Understanding Memory Mapping in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

ftw or ftw64 Subroutine

Purpose

Walks a file tree.

Library

Standard C Library (libc.a)

Syntax

```
#include <ftw.h>
int ftw ( Path, Function, Depth)
char *Path;
int (*Function(const char*, const struct stat*, int);
int Depth;
int ftw64 ( Path, Function, Depth)
char *Path;
int (*Function(const char*, const struct stat64*, int);
int Depth;
```

Description

The ftw and ftw64 subroutines recursively searches the directory hierarchy that descends from the directory specified by the *Path* parameter.

For each file in the hierarchy, the ftw and ftw64 subroutines call the function specified by the Function parameter. ftw passes it a pointer to a null-terminated character string containing the name of the file, a pointer to a stat structure containing information about the file, and an integer. ftw64 passes it a pointer to a null-terminated character string containing the name of the file, a pointer to a stat64 structure containing information about the file, and an integer.

The integer passed to the *Function* parameter identifies the file type with one of the following values:

FTW F Regular file FTW_D Directory

FTW_DNR Directory that cannot be read

FTW_SL Symbolic Link

FTW_NS File for which the stat structure could not be executed successfully

If the integer is FTW-DNR, the files and subdirectories contained in that directory are not processed.

If the integer is FTW-NS, the stat structure contents are meaningless. An example of a file that causes FTW-NS to be passed to the Function parameter is a file in a directory for which you have read permission but not execute (search) permission.

The ftw and ftw64 subroutines finish processing a directory before processing any of its files or subdirectories.

The ftw and ftw64 subroutines continue the search until the directory hierarchy specified by the Path parameter is completed, an invocation of the function specified by the *Function* parameter returns a nonzero value, or an error is detected within the ftw and ftw64 subroutines, such as an I/O error.

The ftw and ftw64 subroutines traverse symbolic links encountered in the resolution of the Path parameter, including the final component. Symbolic links encountered while walking the directory tree rooted at the *Path* parameter are not traversed.

The ftw and ftw64 subroutines use one file descriptor for each level in the tree. The Depth parameter specifies the maximum number of file descriptors to be used. In general, the ftw and ftw64 subroutines runs faster if the value of the *Depth* parameter is at least as large as the number of levels in the tree. However, the value of the *Depth* parameter must not be greater than the number of file descriptors currently available for use. If the value of the *Depth* parameter is 0 or a negative number, the effect is the same as if it were 1.

Because the ftw and ftw64 subroutines are recursive, it is possible for it to terminate with a memory fault due to stack overflow when applied to very deep file structures.

The ftw and ftw64 subroutines use the malloc subroutine to allocate dynamic storage during its operation. If the ftw and ftw64 subroutined is terminated prior to its completion, such as by the longimp subroutine being executed by the function specified by the Function parameter or by an interrupt routine, the ftw and ftw64 subroutines cannot free that storage. The storage remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the Function parameter return a nonzero value the next time it is called.

Parameters

Path Specifies the directory hierarchy to be searched.

Function Specifies the file type.

Specifies the maximum number of file descriptors to be used. Depth cannot be greater than Depth

OPEN_MAX which is described in the sys/limits.h header file.

Return Values

If the tree is exhausted, the **ftw** and **ftw64** subroutines returns a value of 0. If the subroutine pointed to by fn returns a nonzero value, ftw and ftw64 subroutines stops its tree traversal and returns whatever value was returned by the subroutine pointed to by fn. If the ftw and ftw64 subroutines detects an error, it returns a -1 and sets the errno global variable to indicate the error.

Error Codes

If the ftw or ftw64 subroutines detect an error, a value of -1 is returned and the errno global variable is set to indicate the error.

The ftw and ftw64 subroutine are unsuccessful if:

EACCES Search permission is denied for any component of the Path parameter or read

permission is denied for Path.

ENAMETOOLONG The length of the path exceeds PATH_MAX while _POSIX_NO_TRUNC is in effect.

ENOENT The Path parameter points to the name of a file that does not exist or points to an empty

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

The **ftw** subroutine is unsuccessful if:

EOVERFLOW A file in Path is of a size larger than 2 Gigabytes.

Related Information

The malloc, free, realloc, calloc, mallopt, mallinfo, or alloca ("malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, or valloc Subroutine" on page 657) subroutine, setimp or longimp subroutine, signal subroutine, stat subroutine.

Searching and Sorting Example Program and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

fwide Subroutine

Purpose

Set stream orientation.

Library

Standard Library (libc.a)

Syntax

```
#include <stdio.h>
#include <wchar.h>
int fwid (FILE * stream, int mode),
```

Description

The fwide function determines the orientation of the stream pointed to by stream. If mode is greater than zero, the function first attempts to make the stream wide-oriented. If mode is less than zero, the function first attempts to make the stream byte-oriented. Otherwise, mode is zero and the function does not alter the orientation of the stream.

If the orientation of the stream has already been determined, fwide does not change it.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set errno to 0, then call fwide, then check errno and if it is non-zero, assume an error has occurred.

A call to **fwide** with mode set to zero can be used to determine the current orientation of a stream.

Return Values

The fwide function returns a value greater than zero if, after the call, the stream has wide-orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no orientation.

Errors

The fwide function may fail if:

EBADF

The stream argument is not a valid stream.

Related Information

The wchar.h file

fwprintf, wprintf, swprintf Subroutines

Purpose

Print formatted wide-character output.

Library

Standard Library (libc.a)

Syntax

```
#include <stdio.h>
#include <wchar.h>
int fwprintf ( FILE * stream, const wchar t * format, . . .)
int wprintf (const wchar_t * format, . .)
int swprintf (wchar_t *s, size_t n, const wchar_t * format, . . .)
```

Description

The **fwprintf** function places output on the named output stream. The **wprintf** function places output on the standard output stream stdout. The swprintf function places output followed by the null wide-character in consecutive wide-characters starting at *s; no more than n wide-characters are written, including a terminating null wide-character, which is always added (unless **n** is zero).

Each of these functions converts, formats and prints its arguments under control of the format wide-character string. The **format** is composed of zero or more directives: **ordinary wide-characters**, which are simply copied to the output stream and conversion specifications, each of which results in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

EX Conversions can be applied to the **nth** argument after the **format** in the argument list, rather than to the next unused argument. In this case, the conversion wide-character % (see below) is replaced by the sequence %n\$, where n is a decimal integer in the range [1, {NL_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of format wide-character strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

In format wide-character strings containing the %n\$ form of conversion specifications, numbered arguments in the argument list can be referenced from the format wide-character string as many times as required.

In format wide-character strings containing the % form of conversion specifications, each argument in the argument list is used exactly once.

All forms of the **fwprintf** functions allow for the insertion of a language-dependent radix character in the output string, output as a wide-character value. The radix character is defined in the program's locale (category LC NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

EX Each conversion specification is introduced by the % wide-character or by the wide-character sequence %n\$,after which the following appear in sequence:

- Zero or more flags (in any order), which modify the meaning of the conversion specification.
- An optional minimum field width. If the converted value has fewer wide-characters than the field width, it will be padded with spaces by default on the left; it will be padded on the right, if the left-adjustment flag (-), described below, is given to the field width. The field width takes the form of an asterisk (*), described below, or a decimal integer.
- An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x and X conversions; the number of digits to appear after the radix character for the e, E and f conversions; the maximum number of significant digits for the g and G conversions; or the maximum number of wide-characters to be printed from a string in s conversions. The precision takes the form of a period (.) followed either by an asterisk (*), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion wide-character, the behaviour is undefined.
- An optional I (ell) specifying that a following c conversion wide-character applies to a wint_t argument; an optional I specifying that a following s conversion wide-character applies to a wchar_t argument; an

optional h specifying that a following d, i, o, u, x or X conversion wide-character applies to a type short int or type unsigned short int argument (the argument will have been promoted according to the integral promotions, and its value will be converted to type short int or unsigned short int before printing); an optional h specifying that a following n conversion wide-character applies to a pointer to a type short int argument; an optional I (ell) specifying that a following d, i, o, u, x or X conversion wide-character applies to a type long int or unsigned long int argument; an optional I (ell) specifying that a following n conversion wide-character applies to a pointer to a type long int argument; or an optional L specifying that a following e, E, f, g or G conversion wide-character applies to a type long double argument. If an h, I or L appears with any other conversion wide-character, the behavior is undefined.

A **conversion wide-character** that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk (*). In this case an argument of type int supplies the field width or precision. Arguments specifying field width, or precision, or both must appear in that order before the argument, if any, to be converted. A negative field width is taken as a - flag followed by a positive field width. A negative precision is taken as if EX the precision were omitted. In format wide-character strings containing the %n\$ form of a conversion specification, a field width or precision may be indicated by the sequence *m\$, where m is a decimal integer in the range [1, {NL_ARGMAX}] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
wprintf(L"1$d:2$.*3$d:4$.*3$d\n", hour, min, precision, sec);
```

The format can contain either numbered argument specifications (that is, %n\$ and *m\$), or unnumbered argument specifications (that is, % and *), but normally not both. The only exception to this is that %% can be mixed with the **%n\$** form. The results of mixing numbered and unnumbered argument specifications in a format wide-character string are undefined. When numbered argument specifications are used, specifying the Nth argument requires that all the leading arguments, from the first to the (N-1)th, are specified in the format wide-character string.

The flag wide-characters and their meanings are:

- The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %g or %G) will be formatted with thousands' grouping wide-characters. For other conversions the behaviour is undefined. The non-monetary grouping wide-character is used.
- The result of the conversion will be left-justified within the field. The conversion will be right-justified if this flag is not specified.
- The result of a signed conversion will always begin with a sign (+ or -). The conversion will begin with a sign only when a negative value is converted if this flag is not specified.
- If the first wide-character of a signed conversion is not a sign or if a signed conversion results in no space wide-characters, a space will be prefixed to the result. This means that if the space and + flags both appear, the space flag will be ignored.
- This flag specifies that the value is to be converted to an alternative form. For o conversion, it increases the precision (if necessary) to force the first digit of the result to be 0. For x or X conversions, a non-zero result will have 0x (or 0X) prefixed to it. For e, E, f, g or G conversions, the result will always contain a radix character, even if no digits follow it. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversions, trailing zeros will not be removed from the result as they normally are. For other conversions, the behavior is undefined.
- 0 For d, i, o, u, x, X, e, E, f, g and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and - flags both appear, the 0 flag will be ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag will be ignored. If the 0 and 'flags both appear, the grouping wide-characters are inserted before zero padding. For other conversions, the behavior is undefined.

The conversion wide-characters and their meanings are:

- d,i The int argument is converted to a signed decimal in the style [-] dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.
- The unsigned int argument is converted to unsigned octal format in the style dddd. The precision 0 specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.
- u The unsigned int argument is converted to unsigned decimal format in the style dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.
- X The unsigned int argument is converted to unsigned hexadecimal format in the style dddd; the letters abcdef are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.
- Χ Behaves the same as the x conversion wide-character except that letters ABCDEF are used instead of abcdef.
- f The double argument is converted to decimal notation in the style [-] ddd.ddd, where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly 0 and no # flag is present, no radix character appears. If a radix character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

The fwprintf family of functions may make available wide-character string representations for infinity and NaN.

- The double argument is converted in the style [-] d.ddde +/- dd, where there is one digit before the e, E radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is 0 and no # flag is present, no radix character appears. The value is rounded to the appropriate number of digits. The E conversion wide-character will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits. If the value is 0, the exponent is 0.
 - The fwprintf family of functions may make available wide-character string representations for infinity and NaN.
- g, G The double argument is converted in the style f or e (or in the style E in the case of a G conversion wide-character), with the precision specifying the number of significant digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value converted; style e (or E) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a radix character appears only if it is followed by a digit.

The fwprintf family of functions may make available wide-character string representations for infinity

- If no I (ell) qualifier is present, the int argument is converted to a wide-character as if by calling the btowc function and the resulting wide-character is written. Otherwise the wint_t argument is converted to wchar_t, and written.
 - If no I (ell) qualifier is present, the argument must be a pointer to a character array containing a character sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the mbrtowc function, with the conversion state described by an mbstate_t object initialised to zero before the first character is converted, and written up to (but not including) the terminating null wide-character. If the precision is specified, no more than that many wide-characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null wide-character.

If an I (ell) qualifier is present, the argument must be a pointer to an array of type wchar_t. Wide characters from the array are written up to (but not including) a terminating null wide-character. If no precision is specified or is greater than the size of the array, the array must contain a null wide-character. If a precision is specified, no more than that many wide-characters are written.

С

S

The argument must be a pointer to void. The value of the pointer is converted to a sequence of p printable wide-characters, in an implementation-dependent manner. The argument must be a pointer to an integer into which is written the number of wide-characters written to the output so far by this call to one of the fwprintf functions. No argument is converted.

C Same as lc. S Same as Is.

Output a % wide-character; no argument is converted. The entire conversion specification must be %%.

If a conversion specification does not match one of the above forms, the behavior is undefined.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by fwprintf and wprintf are printed as if fputwc had been called.

The st_ctime and st_mtime fields of the file will be marked for update between the call to a successful execution of fwprintf or wprintf and the next successful completion of a call to fflush or fclose on the same stream or a call to exit or abort.

Return Values

Upon successful completion, these functions return the number of wide-characters transmitted excluding the terminating null wide-character in the case of swprintf or a negative value if an output error was encountered.

Error Codes

For the conditions under which fwprintf and wprintf will fail and may fail, refer to fputwc .In addition, all forms of fwprintf may fail if:

EILSEQ A wide-character code that does not correspond to a valid character has been detected

EINVAL There are insufficient arguments.

In addition, wprintf and fwprintf may fail if:

ENOMEM Insufficient storage space is available.

Examples

To print the language-independent date and time format, the following statement could be used:

```
wprintf (format, weekday, month, day, hour, min);
```

For American usage, format could be a pointer to the wide-character string:

```
L"%s, %s %d, %d:%.2d\n"
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, format could be a pointer to the wide-character string:

```
L"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

producing the message:

Sonntag, 3. July, 10:02

Related Information

The **btowc** ("btowc Subroutine" on page 103) subroutine, **fputwc** ("putwc, putwchar, or fputwc Subroutine" on page 995) subroutine, **fwscanf** ("fwscanf, wscanf, swscanf Subroutines") subroutine, **setlocale** subroutine, **mbrtowc** ("mbrtowc Subroutine" on page 669) subroutine.

The wchar.h file.

fwscanf, wscanf, swscanf Subroutines

Purpose

Convert formatted wide-character input.

Library

Standard Library (libc.a)

Syntax

```
#include <stdio.h>
#include <wchar.h>
int fwscanf (FILE * stream, const wchar_t * format, ...);
int wscanf (const wchar_t * format, ...);
int swscanf (const wchar t * s, const wchar t * format, ...);
```

Description

The **fwscanf** function reads from the named input stream. The **wscanf** function reads from the standard input stream stdin. The **swscanf** function reads from the wide-character string s. Each function reads wide-characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control wide-character string format described below, and a set of pointer arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the **nth** argument after the **format** in the argument list, rather than to the next unused argument. In this case, the conversion wide-character % (see below) is replaced by the sequence **%n\$**, where **n** is a decimal integer in the range [1, {NL_ARGMAX}]. This feature provides for the definition of format wide-character strings that select arguments in an order appropriate to specific languages. In format wide-character strings containing the **%n\$** form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format wide-character string more than once.

The format can contain either form of a conversion specification, that is, % or **%n\$**, but the two forms cannot normally be mixed within a single format wide-character string. The only exception to this is that %% or %* can be mixed with the **%n\$** form.

The **fwscanf** function in all its forms allows for detection of a language-dependent radix character in the input string, encoded as a wide-character value. The radix character is defined in the program's locale (category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

The format is a wide-character string composed of zero or more directives. Each directive is composed of one of the following: one or more white-space wide-characters (space, tab, newline, vertical-tab or form-feed characters); an ordinary wide-character (neither % nor a white-space character); or a conversion specification. Each conversion specification is introduced by a % or the sequence **%n\$** after which the following appear in sequence:

- An optional assignment-suppressing character *.
- An optional non-zero decimal integer that specifies the maximum field width.
- · An optional size modifier h, I (ell) or L indicating the size of the receiving object. The conversion wide-characters c, s and [must be preceded by I (ell) if the corresponding argument is a pointer to wchar_t rather than a pointer to a character type. The conversion wide-characters d, i and n must be preceded by h if the corresponding argument is a pointer to **short int** rather than a pointer to **int**, or by I (ell) if it is a pointer to long int. Similarly, the conversion wide-characters o, u and x must be preceded by h if the corresponding argument is a pointer to unsigned short int rather than a pointer to unsigned int, or by I (ell) if it is a pointer to unsigned long int. The conversion wide-characters e, f and g must be preceded by I (ell) if the corresponding argument is a pointer to **double** rather than a pointer to float, or by L if it is a pointer to long double. If an h, I (ell) or L appears with any other conversion wide-character, the behavior is undefined.
- A conversion wide-character that specifies the type of conversion to be applied. The valid conversion wide-characters are described below.

The fwscanf functions execute each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space wide-characters is executed by reading input until no more valid input can be read, or up to the first wide-character which is not a white-space wide-character, which remains unread.

A directive that is an ordinary wide-character is executed as follows. The next wide-character is read from the input and compared with the wide-character that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent wide-characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion wide-character. A conversion specification is executed in the following steps:

Input white-space wide-characters (as specified by iswspace) are skipped, unless the conversion specification includes a [, c or n conversion character.

An item is read from the input, unless the conversion specification includes an n conversion wide-character. An input item is defined as the longest sequence of input wide-characters, not exceeding any specified field width, which is an initial subsequence of a matching sequence. The first wide-character, if any, after the input item remains unread. If the length of the input item is 0, the execution of the conversion specification fails; this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a % conversion wide-character, the input item (or, in the case of a %n conversion specification, the count of input wide-characters) is converted to a type appropriate to the conversion wide-character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result if the conversion specification is introduced by %, or in the nth argument if introduced by the wide-character sequence %n\$. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion wide-characters are valid:

i

O

u

s

[

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of **wcstol** with the value 10 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **int**.

Matches an optionally signed integer, whose format is the same as expected for the subject sequence of **wcstol** with 0 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **int**.

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of **wcstoul** with the value 8 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **unsigned int**.

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of **wcstoul** with the value 10 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **unsigned int**.

Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of wcstoul with the value 16 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to unsigned int.

e, f, gMatches an optionally signed floating-point number, whose format is the same as expected for the subject sequence of **wcstod** . In the absence of a size modifier, the corresponding argument must be a pointer to float.

If the **fwprintf** family of functions generates character string representations for infinity and NaN (a 7858 symbolic entity encoded in floating-point format) to support the ANSI/IEEE Std 754:1985 standard, the **fwscanf**5 family of functions will recognise them as input.

Matches a sequence of non white-space wide-characters. If no I (ell) qualifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically.

Otherwise, the corresponding argument must be a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

Matches a non-empty sequence of wide-characters from a set of expected wide-characters (the **scanset**). If no I (ell) qualifier is present, wide-characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically.

If an I (ell) qualifier is present, the corresponding argument must be a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide-character, which will be added automatically

The conversion specification includes all subsequent wide characters in the **format** string up to and including the matching right square bracket (]). The wide-characters between the square brackets (the **scanlist**) comprise the scanset, unless the wide-character after the left square bracket is a circumflex (^), in which case the scanset contains all wide-characters that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with [] or [^], the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise the first right square bracket is the one that ends the conversion specification. If a - is in the scanlist and is not the first wide-character, nor the second where the first wide-character is a ^;, nor the last wide-character, the behavior is implementation-dependent.

Matches a sequence of wide-characters of the number specified by the field width (1 if no field width is present in the conversion specification). If no I (ell) qualifier is present, wide-characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence. No null character is added.

Otherwise, the corresponding argument must be a pointer to an array of **wchar_t** large enough to accept the sequence. No null wide-character is added.

C

Matches an implementation-dependent set of sequences, which must be the same as the set of p sequences that is produced by the %p conversion of the corresponding fwprintf functions. The corresponding argument must be a pointer to a pointer to void. The interpretation of the input item is implementation-dependent. If the input item is a value converted earlier during the same program execution, the pointer that results will compare equal to that value; otherwise the behavior of the %p conversion is undefined.

No input is consumed. The corresponding argument must be a pointer to the integer into which is to be n written the number of wide-characters read from the input so far by this call to the fwscanf functions. Execution of a %n conversion specification does not increment the assignment count returned at the completion of execution of the function.

C Same as lc. S Same as Is.

Matches a single %; no conversion or assignment occurs. The complete conversion specification must

If a conversion specification is invalid, the behavior is undefined.

The conversion characters E. G and X are also valid and behave the same as, respectively, e. g and x.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any wide-characters matching the current conversion specification (except for %n) have been read (other than leading white-space, where permitted), execution of the current conversion specification terminates with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) is terminated with an input failure.

Reaching the end of the string in swscanf is equivalent to encountering end-of-file for fwscanf.

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white space (including newline) is left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the %n conversion specification.

The fwscanf and wscanf functions may mark the st atime field of the file associated with stream for update. The st atime field will be marked for update by the first successful execution of faetc. faetwc. fgets, fgetws, fread, getc, getwc, getchar, getwchar, gets, fscanf or fwscanf using stream that returns data not supplied by a prior call to **ungetc**.

In format strings containing the % form of conversion specifications, each argument in the argument list is used exactly once.

Return Values

Upon successful completion, these functions return the number of successfully matched and assigned input items; this number can be 0 in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF is returned. If a read error occurs the error indicator for the stream is set, EOF is returned, and errno is set to indicate the error.

Error Codes

For the conditions under which the **fwscanf** functions will fail and may fail, refer to **fgetwc**. In addition, fwscanf may fail if:

EILSEQ Input byte sequence does not form a valid character.

EINVAL There are insufficient arguments.

Examples

```
The call:
```

```
int i, n; float x; char name[50];
n = wscanf(L"%d%f%s", &i, &x, name);
with the input line:
25 54.32E-1 Hamster
```

will assign to n the value 3, to i the value 25, to x the value 5.432, and name will contain the string Hamster.

The call:

```
int i; float x; char name[50];
  (void) wscanf(L"%2d%f%*d %[0123456789]", &i, &x, name);
with input:
  56789 0123 56a72
```

will assign 56 to i, 789.0 to x, skip 0123, and place the string 56\0 in **name**. The next call to **getchar** will return the character a.

Related Information

The **getwc** ("getwc, fgetwc, or getwchar Subroutine" on page 397) subroutine, **fwprintf** ("fwprintf, wprintf, swprintf Subroutines" on page 281) subroutine, **setlocale** subroutine, **wcstod** subroutine, **wcstol** subroutine, **wcstoul** subroutine, **wctomb** subroutine.

The wchar.h file.

gai_strerror Subroutine

Purpose

Facilitates consistent error information from EAI_* values returned by the getaddrinfo subroutine.

Library

Library (libc.a)

Syntax

```
#include <sys/socket.h>
#include <netdb.h>
char *
gai_strerror (ecode)
int ecode;
int
gai_strerror_r (ecode, buf, buflen)
int ecode;
char *buf;
int buflen;
```

Description

For multithreaded environments, the second version should be used. In **gai_strerror_r**, *buf* is a pointer to a data area to be filled in. *buflen* is the length (in bytes) available in *buf*.

It is the caller's responsibility to insure that buf is sufficiently large to store the requested information, including a trailing null character. It is the responsibility of the function to insure that no more than buflen bytes are written into buf.

Return Values

If successful, a pointer to a string containing an error message appropriate for the EAI_* errors is returned. If ecode is not one of the EAI_* values, a pointer to a string indicating an unknown error is returned.

Related Information

The getaddrinfo Subroutine, freeaddrinfo Subroutine, and getnameinfo Subroutine articles in AIX 5L Version 5.2 Technical Reference: Communications Volume 2.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

gamma Subroutine

Purpose

Computes the natural logarithm of the gamma function.

Libraries

The gamma: IEEE Math Library (libm.a) or System V Math Library (libmsaa.a)

Syntax

#include <math.h> extern int signgam; double gamma (x)double x;

Description

The **gamma** subroutine computes the logarithm of the gamma function.

The sign of gamma(x) is returned in the external integer **signgam**.

Note: Compile any routine that uses subroutines from the libm.a with the -Im flag. To compile the Igamma.c file, enter:

cc lgamma.c -lm

Parameters

Specifies the value to be computed.

Related Information

"exp. expf. or expl Subroutine" on page 205, "feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, _class, finite, isnan, or unordered Subroutines" on page 138.

The exp, expm1, log, log10, log1p or pow ("exp, expf, or expl Subroutine" on page 205) subroutine, matherr ("matherr Subroutine" on page 665) subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

128-Bit long double Floating-Point Format in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

math.h in AIX 5L Version 5.2 Files Reference.

gencore or coredump Subroutine

Purpose

Creates a core file without terminating the process.

Library

Standard C Library (libc.a)

Syntax

#include <core.h>

int gencore (coredumpinfop) struct coredumpinfo *coredumpinfop;

int coredump (coredumpinfop) struct coredumpinfo *coredumpinfop;

Description

The **gencore** and **coredump** subroutines create a core file of a process without terminating it. The core file contains the snapshot of the process at the time the call is made and can be used with the dbx command for debugging purposes.

If any thread of the process is in a system call when its snapshot core file is generated, the register information returned may not be reliable (except for the stack pointer). To save all user register contents when a system call is made so that they are available to the **gencore** and **coredump** subroutines, the application should be built using the "-bM:UR" flags.

If any thread of the process is sleeping inside the kernel or stopped (possibly for job control), the caller of the **gencore** and **coredump** subroutines will also be blocked until the thread becomes runnable again. Thus, these subroutines may take a long time to complete depending upon the target process state.

The **coredump** subroutine always generates a core file for the process from which it is called. This subroutine has been replaced by the **gencore** subroutine and is being provided for compatibility reasons only.

The **gencore** subroutine creates a core file for the process whose process ID is specified in the *pid* field of the coredumpinfo structure. For security measures, the user ID (uid) and group ID (gid) of the core file are set to the uid and gid of the process.

Both these subroutines return success even if the core file cannot be created completely because of filesystem space constraints. When using the dbx command with an incomplete core file, dbx may warn that the core file is truncated.

In the "Change / Show Characteristics of Operating System" smitty screen, there are two options regarding the creation of the core file. The core file will always be created in the default core format and will ignore the value specified in the "Use pre-430 style CORE dump" option. However, the value

specified for the "Enable full CORE dump" option will be considered when creating the core file. Resource limits of the target process for file and coredump will be enforced.

The **coredumpinfo** structure contains the following fields:

Member Type	Member Name	Description
unsigned int	length	Length of the core file name.
char *	name	Name of the core file.
pid_t	pid	ID of the process to be coredumped.
int	flags	Flags-version flag. Set this to COREGEN_VERSION_1.

Note: The pid and flags fields are required for the gencore subroutine, but are ignored for the coredump subroutine

Parameters

coredumpinfop Specifies the address of the coredumpinfo structure that provides the file name to save

the core snapshot and its length. For the gencore subroutine, it also provides the process id of the process whose core is to be dumped and a flag which includes version

flag bits. The version flag value must be set to COREGEN_VERSION_1.

Return Values

Upon successful completion, the gencore and coredump subroutines return a 0. If unsuccessful, a -1 is returned, and the errno global variable is set to indicate the error

Error Codes

EACCESS Search permission is denied on a component of the path prefix, the file exists and

permissions specified by the mode are denied, or the file does not exist and write

permission is denied for the parent directory of the file to be created.

ENOENT The *name* field in the *coredumpinfo* parameter points to an empty string. **EINTR** The subroutine was interrupted by a signal before it could complete.

ENAMETOOLONG The value of the *length* field in the **coredumpinfop** structure or the length of the

absolute path of the specified core file name is greater than MAXPATHLEN (as defined

in the sys/param.h file).

EINVAL The value of the *length* field in the **coredumpinfop** structure is 0.

The target process is already in the middle of another gencore or coredump **EAGAIN**

subroutine.

ENOMEM Unable to allocate memory resources to complete the subroutine.

In addition to the above, the following errno values can be set when the gencore subroutine is unsuccessful:

EPERM The real or effective user ID of the calling process does

not match the real or effective user ID of target process or the calling process does not have root user authority.

There is no process whose ID matches the value specified **ESRCH**

in the pid field of the coredumpinfop parameter or the

process is exiting.

The flags field in the coredumpinfop parameter is not set **EINVAL**

to a valid version value.

Related Information

The adb Command, in AIX 5L Version 5.2 Commands Reference, Volume 1.

The dbx command, and gencore Command in AIX 5L Version 5.2 Commands Reference, Volume 2.

The core file format in AIX 5L Version 5.2 Files Reference.

get_malloc_log Subroutine

Purpose

Retrieves information about the malloc subsystem.

Syntax

```
#include <malloc.h>
size_t get_malloc_log (addr, buf, bufsize)
void *addr;
void *buf;
size_t bufsize;
```

Description

The **get_malloc_log** subroutine retrieves a record of currently active malloc allocations. These records are stored as an array of **malloc_log** structures, which are copied from the process heap into the buffer specified by the *buf* parameter. No more than *bufsize* bytes are copied into the buffer. Only records corresponding to the heap of which *addr* is a member are copied, unless *addr* is NULL, in which case records from all heaps are copied. The *addr* parameter must be either a pointer to space allocated previously by the malloc subsystem or NULL.

Parameters

addr Pointer to a space allocated by the malloc subsystem.

buf Specifies into which buffer the **malloc_log** structures are stored.
bufsize Specifies the number of bytes that can be copied into the buffer.

Return Values

The **get_malloc_log** subroutine returns the number of bytes actually transferred into the *bufsize* parameter. If Malloc Log is not enabled, 0 is returned. If *addr* is not a pointer allocated by the malloc subsystem, 0 is returned and the **errno** global variable is set to **EINVAL**.

Related Information

"malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, or valloc Subroutine" on page 657, and "get malloc log live Subroutine."

reset_malloc_log Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

get_malloc_log_live Subroutine

Purpose

Provides information about the malloc subsystem.

Syntax

```
#include <malloc.h>
struct malloc_log* get_malloc_log_live (addr)
void *addr;
```

Description

The **get malloc log live** subroutine provides access to a record of currently active malloc allocations. The information is stored as an array of **malloc_log** structures, which are located in the process heap. This data is volatile and subject to update. The addr parameter must be either a pointer to space allocated previously by the malloc subsystem or NULL.

Parameters

addr

Pointer to space allocated previously by the malloc subsystem

Return Values

The get malloc log live subroutine returns a pointer to the process heap at which the records of current malloc allocations are stored. If the addr parameter is NULL, a pointer to the beginning of the array is returned. If addr is a pointer to space allocated previously by the malloc subsystem, the pointer returned corresponds to records of the same heap as addr. If Malloc Log is not enabled, NULL is returned. If addr is not a pointer allocated by the malloc subsystem, NULL is returned and the errno global variable is set to EINVAL.

Related Information

"malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo heap, alloca, or valloc Subroutine" on page 657, and "get malloc log Subroutine" on page 294.

reset_malloc_log Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

get_speed, set_speed, or reset_speed Subroutines

Purpose

Set and get the terminal baud rate.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/str_tty.h>
int get speed (FileDescriptor)
int FileDescriptor;
int set speed (FileDescriptor, Speed)
int FileDescriptor;
int Speed;
int reset_speed (FileDescriptor)
int FileDescriptor;
```

Description

The baud rate functions set speed subroutine and get speed subroutine are provided top allow the user applications to program any value of the baud rate that is supported by the asynchronous adapter, but that cannot be expressed using the termios subroutines cfsetospeed, cfsetispeed, cfgetospeed, and cfsgetispeed. Those subroutines are indeed limited to the set values (BO, B50, ..., B38400) described in <termios.h>.

Interaction with the termios Baud flags:

If the terminal's device driver supports these subroutines, it has two interfaces for baud rate manipulation.

Operation for Baud Rate:

normal mode: This is the default mode, in which a termios supported speed is in use.

speed-extended mode: This mode is entered either by calling set_speed subroutine a non-termios supported speed at the configuration of the line.

In this mode, all the calls to tcgetattr subroutine or TCGETS ioctl subroutine will have B50 in the returned termios structure.

If tcsetatt subroutine or TCSETS, TCSETAF, or TCSETAW ioctl subroutines is called and attempt to set B50, the actual baud rate is not changed. If is attempts to set any other termios-supported speed, the driver will switch back to the normal mode and the requested baud rate is set. Calling reset speed subroutine is another way to switch back to the normal mode.

Parameters

Specifies an open file descriptor. FileDescriptor

Speed The integer value of the requested speed.

Return Values

Upon successful completion, set_speed and reset_speed return a value of 0, and get_speed returns a positive integer specifying the current speed of the line. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

EINVAL

The FileDescriptor parameter does not specify a valid file descriptor for a tty the recognizes the set_speed, get_speed and reset_speed subroutines, or the Speed parameter of set_speed is not supported by the terminal.

Plus all the **errno** codes that may be set in case of failure in an **ioctl** subroutine issued to a streams based tty.

Related Information

cfgetospeed, cfsetospeed, cfgetispeed, or cfsetispeed ("cfgetospeed, cfsetospeed, cfgetispeed, or cfsetispeed Subroutine" on page 118) subroutines.

getargs Subroutine

Purpose

Gets arguments of a process.

Library

Standard C library (libc.a)

Syntax

```
#include cinfo.h>
#include <sys/types.h>
int getargs (processBuffer, bufferLen, argsBuffer, argsLen)
struct procsinfo *processBuffer
or struct procentry64 *processBuffer;
int bufferLen;
char *argsBuffer;
int argsLen;
```

Description

The **getargs** subroutine returns a list of parameters that were passed to a command when it was started. Only one process can be examined per call to **getargs**.

The **getargs** subroutine uses the pi_pid field of *processBuffer* to determine which process to look for. bufferLen should be set to the size of struct procsinfo or struct procentry64. Parameters are returned in argsBuffer, which should be allocated by the caller. The size of this array must be given in argsLen.

On return, argsBuffer consists of a succession of strings, each terminated with a null character (ascii \0'). Hence, two consecutive NULLs indicate the end of the list.

Note: The arguments may be changed asynchronously by the process, but results are not guaranteed to be consistent.

Parameters

processBuffer

Specifies the address of a procsinfo or procentry64 structure, whose pi_pid field should contain the pid of the process that is to be looked for.

bufferLen

Specifies the size of a single **procsinfo** or **procentry64** structure.

argsBuffer

Specifies the address of an array of characters to be filled with a series of strings representing the parameters that are needed. An extra NULL character marks the end of the list. This array must be allocated by the caller.

argsLen

Specifies the size of the argsBuffer array. No more than argsLen characters are returned.

Return Values

If successful, the getargs subroutine returns zero. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **getargs** subroutine does not succeed if the following are true:

ESRCH The specified process does not exist.

EFAULT The copy operation to the buffer was not successful or the processBuffer or

argsBuffer parameters are invalid.

EINVAL

The bufferLen parameter does not contain the size of a single procsinfo or procentry64 structure.

ENOMEM

There is no memory available in the address space.

Related Information

The getevars ("getevars Subroutine" on page 314), getpid ("getpid, getpgrp, or getppid Subroutine" on page 345), getpgrp ("getpid, getpgrp, or getppid Subroutine" on page 345), getppid ("getpid, getpgrp, or getppid Subroutine" on page 345), getprocs or getthrds ("getthrds Subroutine" on page 372) subroutines.

The **ps** command.

getaudithostattr, IDtohost, hosttoID, nexthost or putaudithostattr **Subroutine**

Purpose

Accesses the host information in the audit host database.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
int getaudithostattr (Hostname, Attribute, Value, Type)
char *Hostname;
char *Attribute;
void *Value:
int Type;
char *IDtohost (ID);
char *ID;
char *hosttoID (Hostname, Count);
char *Hostname;
int Count;
char *nexthost (void);
int putaudithostattr (Hostname, Attribute, Value, Type);
char *Hostname;
char *Attribute;
void *Value;
int Type;
```

Description

These subroutines access the audit host information.

The getaudithostattr subroutine reads a specified attribute from the host database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly the putaudithostattr subroutine writes a specified attribute into the host database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the putaudithostattr must be explicitly committed by calling the putaudithostattr subroutine with a Type of SEC COMMIT. Until all the data is committed, only these subroutines within the process return written data.

New entries in the host database must first be created by invoking putaudithostattr with the SEC NEW type.

The **IDtohost** subroutine converts an 8 byte host identifier into a hostname.

The **hosttoID** subroutine converts a hostname to a pointer to an array of valid 8 byte host identifiers. A pointer to the array of identifiers is returned on success. A NULL pointer is returned on failure. The number of known host identifiers is returned in *Count.

The **nexthost** subroutine returns a pointer to the name of the next host in the audit host database.

Parameters

Attribute Specifies which attribute is read. The following possible

attributes are defined in the usersec.h file:

S_AUD_CPUID

Host identifier list. The attribute type is

SEC_LIST.

Count Specifies the number of 8 byte host identifier entries that

are available in the IDarray parameter or that have been

returned in the *IDarray* parameter.

Hostname Specifies the name of the host for the operation.

ID An 8 byte host identifier.

Specifies a pointer to an array of 1 or more 8 byte host **IDarray**

identifiers.

Specifies the type of attribute expected. Valid types are Туре

defined in usersec.h. The only valid Type value is

SEC_LIST.

Value The return value for read operations and the new value for

write operations.

Return Values

On successful completion, the getaudithostattr, IDtohost, hosttoID, nexthost, or putaudithostattr subroutine returns 0. If unsuccessful, the subroutine returns non-zero.

Error Codes

The getaudithostattr, IDtohost, hosttoID, nexthost, or putaudithostattr subroutine fails if the following is true:

EINVAL If invalid attribute Name or if Count is equal to zero for the

hosttoID subroutine.

ENOENT If there is no matching *Hostname* entry in the database.

Related Information

The auditmerge command, auditpr command, auditselect command, auditstream command.

The **auditread** ("auditread, auditread_r Subroutines" on page 92) subroutine.

getauthdb Subroutine

Purpose

Copies the name of the current database module.

Library

Standard C Library (libc.a)

Syntax

int getauthdb (auth_db_name)
char *auth db name;

Description

The **getauthdb** subroutine copies the name of the current database module, if one has been defined by an earlier call to the **setauthdb** subroutine, to the *auth_db_name* parameter. An error is returned by this subroutine if there is no database module defined.

Parameters

auth_db_name	Pointer to where the name of the current module will be	
	stored.	

Return Values

An authentication database module has been specified by an earlier call to the **setauthdb** subroutine. The name of the current database module has been copied to the *auth_db_name* parameter.

An authentication database module has not been

An authentication database module has not been specified by an earlier call to the **setauthdb** subroutine.

Related Information

setauthdb Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

getc, getchar, fgetc, or getw Subroutine

Purpose

Gets a character or word from an input stream.

Library

Standard I/O Package (libc.a)

Syntax

int getc (Stream)
FILE *Stream;
int fgetc (Stream)
FILE *Stream;
int getchar (void)
int getw (Stream)
FILE *Stream;

#include <stdio.h>

Description

The **getc** macro returns the next byte as an **unsigned char** data type converted to an **int** data type from the input specified by the Stream parameter and moves the file pointer, if defined, ahead one byte in the Stream parameter. The getc macro cannot be used where a subroutine is necessary; for example, a subroutine pointer cannot point to it.

Because it is implemented as a macro, the **getc** macro does not work correctly with a *Stream* parameter value that has side effects. In particular, the following does not work: getc(*f++)

In such cases, use the fgetc subroutine.

The **fgetc** subroutine performs the same function as the **getc** macro, but **fgetc** is a true subroutine, not a macro. The fgetc subroutine runs more slowly than getc but takes less disk space.

The getchar macro returns the next byte from stdin (the standard input stream). The getchar macro is equivalent to getc(stdin).

The first successful run of the fgetc, fgets, fgetwc, fgetws, fread, fscanf, getc, getchar, gets or scanf subroutine using a stream that returns data not supplied by a prior call to the ungetc or ungetwo subroutine marks the st atime field for update.

The **getc** and **getchar** macros have also been implemented as subroutines for ANSI compatibility. To access the subroutines instead of the macros, insert #undef getc or #undef getchar at the beginning of the source file.

The **getw** subroutine returns the next word (int) from the input specified by the *Stream* parameter and increments the associated file pointer, if defined, to point to the next word. The size of a word varies from one machine architecture to another. The getw subroutine returns the constant EOF at the end of the file or when an error occurs. Since EOF is a valid integer value, the feof and ferror subroutines should be used to check the success of **getw**. The **getw** subroutine assumes no special alignment in the file.

Because of additional differences in word length and byte ordering from one machine architecture to another, files written using the putw subroutine are machine-dependent and may not be readable using the getw macro on a different type of processor.

Parameters

Stream Points to the file structure of an open file.

Return Values

Upon successful completion, the getc, fgetc, getchar, and getw subroutines return the next byte or int data type from the input stream pointed by the Stream parameter. If the stream is at the end of the file, an end-of-file indicator is set for the stream and the integer constant EOF is returned. If a read error occurs, the errno global variable is set to reflect the error, and a value of EOF is returned. The ferror and feof subroutines should be used to distinguish between the end of the file and an error condition.

Error Codes

If the stream specified by the Stream parameter is unbuffered or data needs to be read into the stream's buffer, the getc, getchar, fgetc, or getw subroutine is unsuccessful under the following error conditions:

EAGAIN Indicates that the O_NONBLOCK flag is set for the file descriptor underlying the stream

specified by the Stream parameter. The process would be delayed in the fgetc subroutine

operation.

EBADF Indicates that the file descriptor underlying the stream specified by the Stream parameter is

not a valid file descriptor opened for reading.

EFBIG Indicates that an attempt was made to read a file that exceeds the process' file-size limit or

the maximum file size. See the ulimit subroutine.

EINTR Indicates that the read operation was terminated due to the receipt of a signal, and either no

> data was transferred, or the implementation does not report partial transfer for this file. Note: Depending upon which library routine the application binds to, this subroutine may

return **EINTR**. Refer to the **signal** subroutine regarding **sa_restart**.

EIO Indicates that a physical error has occurred, or the process is in a background process group

> attempting to perform a read subroutine call from its controlling terminal, and either the process is ignoring (or blocking) the SIGTTIN signal or the process group is orphaned.

EPIPE Indicates that an attempt is made to read from a pipe or first-in-first-out (FIFO) that is not

open for reading by any process. A SIGPIPE signal will also be sent to the process.

EOVERFLOW Indicates that the file is a regular file and an attempt was made to read at or beyond the

offset maximum associated with the corresponding stream.

The **getc**, **getchar**, **fgetc**, or **getw** subroutine is also unsuccessful under the following error conditions:

ENOMEM Indicates insufficient storage space is available.

ENXIO Indicates either a request was made of a nonexistent device or the request was outside the

capabilities of the device.

Related Information

The feof, ferror, clearerr, or fileno ("feof, ferror, clearerr, or fileno Macro" on page 227) subroutine, freopen, fopen, or fdopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245)subroutine, fread or fwrite ("fread or fwrite Subroutine" on page 267) subroutine, getwc, fgetwc, or getwchar ("getwc, fgetwc, or getwchar Subroutine" on page 397)subroutine, get or fgets ("gets or fgets Subroutine" on page 366) subroutine, putc, putchar, fputc, or putw ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine, scanf, sscanf, fscanf, or wsscanf subroutine.

List of Character Manipulation Services, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getc unlocked, getchar unlocked, putc unlocked, putchar unlocked **Subroutines**

Purpose

stdio with explicit client locking.

Library

Standard Library (libc.a)

Syntax

```
#include <stdio.h>
int getc unlocked (FILE * stream);
int getchar unlocked (void):
int putc unlocked (int c, FILE * stream);
int putchar_unlocked (int c);
```

Description

Versions of the functions getc, getchar, putc, and putchar respectively named getc_unlocked, getchar_unlocked, putc_unlocked, and putchar_unlocked are provided which are functionally identical to the original versions with the exception that they are not required to be implemented in a thread-safe manner. They may only safely be used within a scope protected by flockfile (or ftrylockfile) and funlockfile. These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE*) object, as is the case after a successful call of the flockfile or ftrylockfile functions.

Return Values

See getc, getchar, putc, and putchar.

Related Information

The getc or getchar ("getc, getchar, fgetc, or getw Subroutine" on page 300) subroutine, putc or putchar ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine.

getconfattr Subroutine

Purpose

Accesses the system information in the user database.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
#include <userconf.h>
int getconfattr (sys, Attribute, Value, Type)
char * sys;
char * Attribute;
void *Value;
int Type;
int putconfattr (sys, Attribute, Value, Type)
char * sys;
char * Attribute;
void *Value;
int Type;
```

Description

The getconfattr subroutine reads a specified attribute from the user database. The putconfattr subroutine writes a specified attribute to the user database.

Parameters

System attribute. The following possible attributes are defined in the userconf.h file. sys

- SC_SYS_LOGIN
- SC_SYS_USER
- SC SYS ADMUSER
- · SC SYS AUDIT SEC LIST

- SC_SYS_AUSERS SEC_LIST
- SC SYS_ASYS SEC_LIST
- SC_SYS_ABIN SEC_LIST
- SC_SYS_ASTREAM SEC_LIST

Attribute

Specifies which attribute is read. The following possible attributes are defined in the usersec.h file:

S_ID User ID. The attribute type is **SEC_INT**.

S PGRP

Principle group name. The attribute type is **SEC_CHAR**.

S_GROUPS

Groups to which the user belongs. The attribute type is **SEC_LIST**.

S ADMGROUPS

Groups for which the user is an administrator. The attribute type is **SEC_LIST**.

S ADMIN

Administrative status of a user. The attribute type is **SEC_BOOL**.

S AUDITCLASSES

Audit classes to which the user belongs. The attribute type is **SEC_LIST**.

S AUTHSYSTEM

Defines the user's authentication method. The attribute type is **SEC_CHAR.**

S HOME

Home directory. The attribute type is **SEC_CHAR**.

S SHELL

Initial program run by a user. The attribute type is **SEC_CHAR**.

S_GECOS

Personal information for a user. The attribute type is **SEC_CHAR**.

S USRENV

User-state environment variables. The attribute type is **SEC_LIST**.

S SYSENV

Protected-state environment variables. The attribute type is **SEC_LIST**.

S LOGINCHK

Specifies whether the user account can be used for local logins. The attribute type is **SEC_BOOL**.

S HISTEXPIRE

Defines the period of time (in weeks) that a user cannot reuse a password. The attribute type is **SEC_INT**.

S HISTSIZE

Specifies the number of previous passwords that the user cannot reuse. The attribute type is **SEC INT**.

S MAXREPEAT

Defines the maximum number of times a user can repeat a character in a new password. The attribute type is **SEC_INT**.

S MINAGE

Defines the minimum age in weeks that the user's password must exist before the user can change it. The attribute type is **SEC INT**.

S PWDCHECKS

Defines the password restriction methods for this account. The attribute type is **SEC LIST**.

S MINALPHA

Defines the minimum number of alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S_MINDIFF

Defines the minimum number of characters required in a new password that were not in the old password. The attribute type is **SEC_INT**.

S MINLEN

Defines the minimum length of a user's password. The attribute type is **SEC_INT**.

S MINOTHER

Defines the minimum number of non-alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S DICTIONLIST

Defines the password dictionaries for this account. The attribute type is SEC_LIST.

S SUCHK

Specifies whether the user account can be accessed with the **su** command. Type SEC BOOL.

S REGISTRY

Defines the user's authentication registry. The attribute type is **SEC_CHAR**.

S RLOGINCHK

Specifies whether the user account can be used for remote logins using the telnet or rlogin commands. The attribute type is SEC BOOL.

S DAEMONCHK

Specifies whether the user account can be used for daemon execution of programs and subsystems using the cron daemon or src. The attribute type is SEC BOOL.

S TPATH

Defines how the account may be used on the trusted path. The attribute type is **SEC_CHAR**. This attribute must be one of the following values:

nosak The secure attention key is not enabled for this account.

notsh The trusted shell cannot be accessed from this account.

always

This account may only run trusted programs.

Normal trusted-path processing applies. on

S TTYS

List of ttys that can or cannot be used to access this account. The attribute type is SEC_LIST.

S_SUGROUPS

Groups that can or cannot access this account. The attribute type is **SEC LIST**.

Expiration date for this account, in seconds since the epoch. The attribute type is SEC CHAR.

S AUTH1

Primary authentication methods for this account. The attribute type is SEC_LIST.

S AUTH2

Secondary authentication methods for this account. The attribute type is SEC_LIST.

S UFSIZE

Process file size soft limit. The attribute type is **SEC_INT**.

S UCPU

Process CPU time soft limit. The attribute type is **SEC_INT**.

S UDATA

Process data segment size soft limit. The attribute type is **SEC_INT**.

S USTACK

Process stack segment size soft limit. Type: **SEC_INT**.

S URSS

Process real memory size soft limit. Type: **SEC_INT**.

S UCORE

Process core file size soft limit. The attribute type is **SEC_INT**.

S PWD

Specifies the value of the passwd field in the **/etc/passwd** file. The attribute type is **SEC_CHAR**.

S UMASK

File creation mask for a user. The attribute type is **SEC_INT**.

S LOCKED

Specifies whether the user's account can be logged into. The attribute type is **SEC_BOOL**.

S UFSIZE HARD

Process file size hard limit. The attribute type is **SEC_INT**.

S UCPU HARD

Process CPU time hard limit. The attribute type is **SEC_INT**.

S UDATA HARD

Process data segment size hard limit. The attribute type is SEC_INT.

S USTACK HARD

Process stack segment size hard limit. Type: **SEC_INT**.

S_URSS_HARD

Process real memory size hard limit. Type: **SEC_INT**.

S_UCORE_HARD

Process core file size hard limit. The attribute type is **SEC_INT**.

Note: These values are string constants that should be used by applications both for convenience and to permit optimization in latter implementations.

Type Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include:

SEC INT

The format of the attribute is an integer.

For the **getuserattr** subroutine, the user should supply a pointer to a defined integer variable. For the **putuserattr** subroutine, the user should supply an integer.

SEC_CHAR

The format of the attribute is a null-terminated character string.

SEC LIST

The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters.

SEC_BOOL

The format of the attribute from **getuserattr** is an integer with the value of either 0 (false) or 1 (true). The format of the attribute for putuserattr is a null-terminated string containing one of the following strings: true, false, yes, no, always, or never.

SEC COMMIT

For the putuserattr subroutine, this value specified by itself indicates that changes to the named user are to be committed to permanent storage. The Attribute and Value parameters are ignored. If no user is specified, the changes to all modified users are committed to permanent storage.

SEC DELETE

The corresponding attribute is deleted from the database.

SEC NEW

Updates all the user database files with the new user name when using the putuserattr subroutine.

Security

Files Accessed:

File Mode

/etc/security/user rw /etc/security/limits rw /etc/security/login.cfg rw

Return Values

If successful, returns 0

If unsuccessful, returns -1

Error Codes

ENOENT The specified User parameter does not exist or the attribute is not defined for this user.

Files

/etc/passwd Contains user IDs.

Related Information

The getuserattr ("getuserattr, IDtouser, nextuser, or putuserattr Subroutine" on page 382) subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getcontext or setcontext Subroutine

Purpose

Initializes the structure pointed to by *ucp* to the context of the calling process.

Library

(libc.a)

Syntax

#include <ucontext.h>

int getcontext (ucontext_t *ucp);

int setcontext (const uncontext_t *ucp);

Description

The **getcontext** subroutine initalizes the structure pointed to by *ucp* to the current user context of the calling process. The **ucontext_t** type that *ucp* points to defines the user context and includes the contents of the calling process' machine registers, the signal mask, and the current execution stack.

The setcontext subroutine restores the user context pointed to by ucp. A successful call to setcontext subroutine does not return; program execution resumes at the point specified by the ucp argument passed to setcontext subroutine. The ucp argument should be created either by a prior call to getcontext subroutine, or by being passed as an argument to a signal handler. If the ucp argument was created with getcontext subroutine, program execution continues as if the corresponding call of getcontext subroutine had just returned. If the ucp argument was created with makecontext subroutine, program execution continues with the function passed to makecontext subroutine. When that function returns, the process continues as if after a call to setcontext subroutine with the ucp argument that was input to makecontext subroutine. If the ucp argument was passed to a signal handler, program execution continues with the program instruction following the instruction interrupted by the signal. If the uc link member of the ucontext t structure pointed to by the ucp arguement is equal to 0, then this context is the main context, and the process will exit when this context returns.

Parameters

иср

A pointer to a user stucture.

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The getcontext and setcontext subroutines are unsuccessful if one of the following is true:

EINVAL NULL ucp address **EFAULT** Invalid ucp address

Related Information

The makecontext ("makecontext or swapcontext Subroutine" on page 664) subroutine, setjmp subroutine, sigaltstack subroutine, sigaction subroutine, sigprocmask subroutine, and sigsetimp subroutine.

getcwd Subroutine

Purpose

Gets the path name of the current directory.

Library

Standard C Library (libc.a)

Syntax

```
#include <unistd.h>
char *getcwd ( Buffer, Size)
char *Buffer;
size t Size;
```

Description

The **getcwd** subroutine places the absolute path name of the current working directory in the array pointed to by the Buffer parameter, and returns that path name. The size parameter specifies the size in bytes of the character array pointed to by the Buffer parameter.

Parameters

Buffer

Points to string space that will contain the path name. If the Buffer parameter value is a null pointer, the getcwd subroutine, using the malloc subroutine, obtains the number of bytes of free space as specified by the Size parameter. In this case, the pointer returned by the getcwd subroutine can be used as the parameter in a subsequent call to the free subroutine. Starting the getcwd subroutine with a null pointer as the Buffer parameter value is not recommended.

Size

Specifies the length of the string space. The value of the Size parameter must be at least 1 greater than the length of the path name to be returned.

Return Values

If the getcwd subroutine is unsuccessful, a null value is returned and the errno global variable is set to indicate the error. The getcwd subroutine is unsuccessful if the Size parameter is not large enough or if an error occurs in a lower-level function.

Error Codes

If the getcwd subroutine is unsuccessful, it returns one or more of the following error codes:

EACCES Indicates that read or search permission was denied for a component of the path name

EINVAL Indicates that the Size parameter is 0 or a negative number.

ENOMEM Indicates that insufficient storage space is available.

ERANGE Indicates that the Size parameter is greater than 0, but is smaller than the length of the

path name plus 1.

Related Information

The **getwd** ("getwd Subroutine" on page 398) subroutine, **malloc** ("malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, or valloc Subroutine" on page 657) subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getdate Subroutine

Purpose

Convert user format date and time.

Library

Standard C Library (libc.a)

Syntax

```
#include <time.h>
struct tm *getdate (const char *string)
extern int getdate_err
```

Description

The **getdate** subroutine converts user definable date and/or time specifications pointed to by *string*, into a **struct tm**. The structure declaration is in the **time.h** header file (see **ctime** subroutine).

User supplied templates are used to parse and interpret the input string. The templates are contained in text files created by the user and identified by the environment variable **DATEMSK**. The **DATEMSK** variable should be set to indicate the full pathname of the file that contains the templates. The first line in the template that matches the input specification is used for interpretation and conversation into the internal time format.

The templates should follow a format where complex field descriptors are preceded by simpler ones. For example:

through use of algorithms.

```
%M
%H:%M
%m/%d/%y
%m/%d/%y %H:%M:%S
```

The following field descriptors are supported:

%%	Same as %.
%a	Abbreviated weekday name.
% A	Full weekday name.
%b	Abbreviated month name.
%B	Full month name.
%с	Locale's appropriate date and time representation.
%C	Century number (00-99; leading zeros are permitted but not required)
%d	Day of month (01 - 31: the leading zero is optional.
%e	Same as %d.
%D	Date as %m/%d/%y.
%h	Abbreviated month name.
%H	Hour (00 - 23)
%l	Hour (01 - 12)
%m	Month number (01 - 12)
%M	Minute (00 - 59)
%n	Same as \n.
%р	Locale's equivalent of either AM or PM.
%r	Time as %I:%M:%S %p
%R	Time as %H: %M
%S	Seconds (00 - 61) Leap seconds are allowed but are not predictable the
%t	Same as tab.

%T Time as %H: %M:%S

Weekday number (Sunday = 0 - 6) %w Locale's appropriate date representation. %x Locale's appropriate time representation. %X

%у Year within century.

> Note: When the environment variable XPG_TIME_FMT=ON, %y is the year within the century. When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969 to 1999, inclusive); values in the range 00-68 refer to 2000 to 2068,

inclusive.

%Y Year as ccyy (such as 1986)

%Z Time zone name or no characters if no time zone exists. If the time zone supplied by %Z is not

> the same as the time zone getdate subroutine expects, an invalid input specification error will result. The getdate subroutine calculates an expected time zone based on information supplied

to the interface (such as hour, day, and month).

The match between the template and input specification performed by the getdate subroutine is case sensitive.

The month and weekday names can consist of any combination of upper and lower case letters. The used can request that the input date or time specification be in a specific language by setting the LC_TIME category (See the setlocale subroutine).

Leading zero's are not necessary for the descriptors that allow leading zero's. However, at most two digits are allowed for those descriptors, including leading zero's. Extra whitespace in either the template file or in string is ignored.

The field descriptors %c, %x, and %X will not be supported if they include unsupported field descriptors.

Example 1 is an example of a template. Example 2 contains valid input specifications for the template. Example 3 shows how local date and time specifications can be defined in the template.

The following rules apply for converting the input specification into the internal format:

- · If only the weekday is given, today is assumed if the given month is equal to the current day and next week if it is less.
- · If only the month is given, the current month is assumed if the given month is equal to the current month and next year if it is less and no year is given (the first day of month is assumed if no day is given).
- · If no hour, minute, and second are given, the current hour, minute and second are assumed.
- · If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

Return Values

Upon successful completion, the getdate subroutine returns a pointer to struct tm; otherwise, it returns a null pointer and the external variable **getdate err** is set to indicate the error.

Error Codes

Upon failure, a null pointer is returned and the variable getdate err is set to indicate the error.

The following is a complete list of the **getdate err** settings and their corresponding descriptions:

- 1 The **DATEMSK** environment variable is null or undefined.
- 2 The template file cannot be opened for reading.
- 3 Failed to get file status information.

- 4 The template file is not a regular file.
- 5 An error is encountered while reading the template file.
- 6 Memory allocation failed (not enough memory available.
- 7 There is no line in the template that matches the input.
- 8 Invalid input specification, Example: February 31 or a time is specified that can not be represented in a time_t (representing the time in seconds since 00:00:00 UTC, January 1, 1970).

Examples

1. The following example shows the possible contents of a template:

```
%A %B %d, %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d, %m, %Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p, %B %dnd
&A den %d. %B %Y %H.%M Uhr
```

2. The following are examples of valid input specifications for the template in Example 1:

```
getdate ("10/1/87 4 PM")
getdate ("Friday")
getdate ("Friday September 18, 1987, 10:30:30")
getdate ("24,9,1986 10:30")
getdate ("at monday the 1st of december in 1986")
getdate ("run job at 3 PM. december 2nd")
```

If the LC TIME category is set to a German locale that includes freitag as a weekday name and oktober as a month name, the following would be valid:

```
getdate ("freitag den 10. oktober 1986 10.30 Uhr")
```

3. The following examples shows how local date and time specification can be defined in the template.

Invocation	Line in Template
getdate ("11/27/86")	%m/%d/%y
getdate ("27.11.86"0	%d.%m.%y
getdate ("86-11-27")	%y-%m-%d
getdate ("Friday 12:00:00")	%A %H:%M:%S

4. The following examples help to illustrate the above rules assuming that the current date Mon Sep 22 12:19:47 EDT 1986 and the LC_TIME category is set to the default "C" locale.

Input	Line in Template	Date
Mon	%a	Mon Sep 22 12:19:47 EDT 1986
Sun	%a	Sun Sep 28 12:19:47 EDT 1986
Fri	%a	Fri Sep 26 12:19:47 EDT 1986
September	%B	Mon Sep1 12:19:47 EDT 1986
January	%B	Thu Jan 1 12:19:47 EDT 1986
December	%B	Mon Dec 1 12:19:47 EDT 1986
Sep Mon	%b %a	Mon Sep 1 12:19:47 EDT 1986
Jan Fri	%b %a	Fri Jan 2 12:19:47 EDT 1986
Dec Mon	%b %a	Mon Dec 1 12:19:47 EDT 1986
Jan Wed 1989	%b %a %Y	Wed Jan 4 12:19:47 EDT 1986

Input	Line in Template	Date
Fri 9	%a %H	Fri Sep 26 12:19:47 EDT 1986
Feb 10:30	%b %H: %S	Sun Feb 1 12:19:47 EDT 1986
10:30	%H: %M	Tue Sep 23 12:19:47 EDT 1986
13:30	%H: %M	Mon Sep 22 12:19:47 EDT 1986

Related Information

The ctime ("ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine" on page 163), ctype ("ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines" on page 168), setlocale, strftime, and times ("getrusage, getrusage64, times, or vtimes Subroutine" on page 360) subroutines.

getdtablesize Subroutine

Purpose

Gets the descriptor table size.

Library

Standard C Library (libc.a)

Syntax

#include <unistd.h>

int getdtablesize (void)

Description

The getdtablesize subroutine is used to determine the size of the file descriptor table.

The size of the file descriptor table for a process is set by the ulimit command or by the setrlimit subroutine. The **getdtablesize** subroutine returns the current size of the table as reported by the **getrlimit** subroutine. If getrlimit reports that the table size is unlimited, getdtablesize instead returns the value of OPEN MAX, which is the largest possible size of the table.

Note: The getdtablesize subroutine returns a runtime value that is specific to the version of the operating system on which the application is running. In AIX 4.3.1, getdtablesize returns a value that is set in the **limits** file, which can be different from system to system.

Return Values

The **getdtablesize** subroutine returns the size of the descriptor table.

Related Information

The close ("close Subroutine" on page 141) subroutine, open ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine, **select** subroutine.

getenv Subroutine

Purpose

Returns the value of an environment variable.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
char *getenv ( Name)
const char *Name;
```

Description

The getenv subroutine searches the environment list for a string of the form Name=Value. Environment variables are sometimes called shell variables because they are frequently set with shell commands.

Parameters

Name

Specifies the name of an environment variable. If a string of the proper form is not present in the current environment, the getenv subroutine returns a null pointer.

Return Values

The **getenv** subroutine returns a pointer to the value in the current environment, if such a string is present. If such a string is not present, a null pointer is returned. The **geteny** subroutine normally does not modify the returned string. The putenv subroutine, however, may overwrite or change the returned string. Do not attempt to free the returned pointer. The **getenv** subroutine returns a pointer to the user's copy of the environment (which is static), until the first invocation of the putenv subroutine that adds a new environment variable. The puterv subroutine allocates an area of memory large enough to hold both the user's environment and the new variable. The next call to the getenv subroutine returns a pointer to this newly allocated space that is not static. Subsequent calls by the **puteny** subroutine use the **realloc** subroutine to make space for new variables. Unsuccessful completion returns a null pointer.

Related Information

The **putenv** ("putenv Subroutine" on page 992) subroutine.

getevars Subroutine

Purpose

Gets environment of a process.

Library

Standard C library (libc.a)

Syntax

```
#include cinfo.h>
#include <sys/types.h>
int getevars (processBuffer, bufferLen, argsBuffer, argsLen)
struct procsinfo *processBuffer
or struct procentry64 *processBuffer;
int bufferLen;
char *argsBuffer;
int argsLen;
```

Description

The getevars subroutine returns the environment that was passed to a command when it was started. Only one process can be examined per call to getevars.

The **getevars** subroutine uses the pi_pid field of *processBuffer* to determine which process to look for. bufferLen should be set to size of struct procsinfo or struct procentry64. Parameters are returned in argsBuffer, which should be allocated by the caller. The size of this array must be given in argsLen.

On return, argsBuffer consists of a succession of strings, each terminated with a null character (ascii \0'). Hence, two consecutive NULLs indicate the end of the list.

Note: The arguments may be changed asynchronously by the process, but results are not guaranteed to be consistent.

Parameters

processBuffer

Specifies the address of a proceinfo or procentry64 structure, whose pi pid field should contain the pid of the process that is to be looked for.

bufferLen

Specifies the size of a single **procsinfo** or **procentry64** structure.

argsBuffer

Specifies the address of an array of characters to be filled with a series of strings representing the parameters that are needed. An extra NULL character marks the end of the list. This array must be allocated by the caller.

argsLen

Specifies the size of the argsBuffer array. No more than argsLen characters are returned.

Return Values

If successful, the getevars subroutine returns zero. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **getevars** subroutine does not succeed if the following are true:

ESRCH The specified process does not exist.

EFAULT The copy operation to the buffer was not successful or the processBuffer or argsBuffer

parameters are invalid.

EINVAL The bufferLen parameter does not contain the size of a single procsinfo or procentry64

ENOMEM There is no memory available in the address space.

Related Information

The getargs ("getargs Subroutine" on page 296), getpid ("getpid, getpgrp, or getppid Subroutine" on page 345), **getpgrp** ("getpid, getpgrp, or getppid Subroutine" on page 345), **getppid** ("getpid, getpgrp, or getppid Subroutine" on page 345), getprocs or getthrds ("getthrds Subroutine" on page 372) subroutines.

The **ps** command.

getfsent, getfsspec, getfsfile, getfstype, setfsent, or endfsent Subroutine

Purpose

Gets information about a file system.

Library

Standard C Library (libc.a)

Syntax

```
#include <fstab.h>
struct fstab *getfsent( )
struct fstab *getfsspec ( Special)
char *Special;
struct fstab *getfsfile( File)
char *File;
struct fstab *getfstype( Type)
char *Type;
void setfsent( )
void endfsent( )
```

Description

The getfsent subroutine reads the next line of the /etc/filesystems file, opening the file if necessary.

The **setfsent** subroutine opens the **/etc/filesystems** file and positions to the first record.

The **endfsent** subroutine closes the **/etc/filesystems** file.

The getfsspec and getfsfile subroutines sequentially search from the beginning of the file until a matching special file name or file-system file name is found, or until the end of the file is encountered. The getfstype subroutine does likewise, matching on the file-system type field.

Note: All information is contained in a static area, which must be copied to be saved.

Parameters

Special Specifies the file-system file name.

File Specifies the file name. Type Specifies the file-system type.

Return Values

The getfsent, getfsspec, getfstype, and getfsfile subroutines return a pointer to a structure that contains information about a file system. The header file fstab.h describes the structure. A null pointer is returned when the end of file (EOF) is reached or if an error occurs.

Files

/etc/filesystems

Centralizes file system characteristics.

Related Information

The getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag, setvfsent, or endvfsent ("getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag, setvfsent, or endvfsent Subroutine" on page 395) subroutine.

The **filesystems** file.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getgid, getegid or gegidx Subroutine

Purpose

Gets the process group IDs.

Library

Standard C Library (libc.a)

Syntax

```
#include <unistd.h>
#include <sys/types.h>
gid t getgid (void);
gid_t getegid (void);
#include <id.h>
gid t getgidx (int type);
```

Description

The **getgid** subroutine returns the real group ID of the calling process.

The **getegid** subroutine returns the effective group ID of the calling process.

The **getgidx** subroutine returns the group ID indicated by the *type* parameter of the calling process.

These subroutines are part of Base Operating System (BOS) Runtime.

Return Values

The getgid, getegidand getgidx subroutines return the requested group ID. The getgid and getegid subroutines are always successful.

The getgidx subroutine will return -1 and set the global errno variable to EINVAL if the type parameter is not one of ID_REAL, ID_EFFECTIVE or ID_SAVED.

Parameters

type

Specifies the group ID to get. Must be one of ID REAL (real group ID), ID EFFECTIVE (effective group ID) or ID_SAVED (saved set-group ID).

Error Codes

If the **getgidx** subroutine fails the following is returned:

EINVAL

Indicates the value of the type parameter is invalid.

Related Information

The getgroups subroutine, initgroups subroutine, setgid subroutine, setgroups subroutine.

The groups command, setgroups command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine

Purpose

Accesses the basic group information in the user database.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrent ( );

struct group *getgrgid (GID)
gid_t GID;

struct group *getgrnam (Name)
const char * Name;
void setgrent ( );
void endgrent ( );
```

Description

Attention: The information returned by the **getgrent**, **getgrnam**, and **getgrgid** subroutines is stored in a static area and is overwritten on subsequent calls. You must copy this information to save it.

Attention: These subroutines should not be used with the **getgroupattr** subroutine. The results are unpredictable.

The **setgrent** subroutine opens the user database if it is not already open. Then, this subroutine sets the cursor to point to the first group entry in the database.

The **getgrent**, **getgrnam**, and **getgrgid** subroutines return information about the requested group. The **getgrent** subroutine returns the next group in the sequential search. The **getgrnam** subroutine returns the first group in the database whose name matches that of the *Name* parameter. The **getgrgid** subroutine returns the first group in the database whose group ID matches the *GID* parameter. The **endgrent** subroutine closes the user database.

Note: An! (exclamation mark) is written into the gr passwd field. This field is ignored and is present only for compatibility with older versions of UNIX.

The Group Structure

The **group** structure is defined in the **grp.h** file and has the following fields:

gr name Contains the name of the group. Contains the password of the group. gr_passwd

Note: This field is no longer used.

gr gid Contains the ID of the group.

Contains the members of the group. gr mem

If the Network Information Service (NIS) is enabled on the system, these subroutines attempt to retrieve the group information from the NIS authentication server.

Parameters

GID Specifies the group ID. Name Specifies the group name.

Group Specifies the basic group information to enter into the user database.

Return Values

If successful, the getgrent, getgrnam, and getgrgid subroutines return a pointer to a valid group structure. Otherwise, a null pointer is returned.

Error Codes

These subroutines fail if one or more of the following are returned:

EIO Indicates that an input/output (I/O) error has occurred.

EINTR Indicates that a signal was caught during the getgrnam or getgrgid subroutine.

EMFILE Indicates that the maximum number of file descriptors specified by the OPEN_MAX value

are currently open in the calling process.

ENFILE Indicates that the maximum allowable number of files is currently open in the system.

To check an application for error situations, set the errno global variable to a value of 0 before calling the getgrgid subroutine. If the errno global variable is set on return, an error occurred.

File

/etc/group Contains basic group attributes.

Related Information

"putgrent Subroutine" on page 993

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getgrgid_r Subroutine

Purpose

Gets a group database entry for a group ID.

Library

Thread-Safe C Library (libc_r.a)

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);
```

Description

The **getgrgid_r** subroutine updates the **group** structure pointed to by *grp* and stores a pointer to that structure at the location pointed to by *result*. The structure contains an entry from the group database with a matching *gid*. Storage referenced by the group structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX} *sysconf* parameter. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

Return Values

Upon successful completion, **getgrgid_r** returns a pointer to a **struct group** with the structure defined in **<grp.h>** with a matching entry if one is found. The **getgrgid_r** function returns a null pointer if either the requested entry was not found, or an error occurred. On error, *errno* will be set to indicate the error.

The return value points to a static area that is overwritten by a subsequent call to the **getgrent**, **getgrgid**, or **getgrnam** subroutine.

If successful, the **getgrgid_r** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The getgrgid_r function fails if:

ERANGE

Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting **group** structure.

Applications wishing to check for error situations should set *errno* to 0 before calling **getgrgid_r**. If *errno* is set on return, an error occurred.

Related Information

The **getgrent**, **getgrgid**, **getgrnam**, **setgrent**, **endgrent** ("getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine" on page 318) subroutine.

The <grp.h>, , and <sys/types.h> header files.

getgrnam_r Subroutine

Purpose

Search a group database for a name.

Library

Thread-Safe C Library (libc_r.a)

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);
```

Description

The **getgrnam_r** function updates the **group** structure pointed to by *grp* and stores pointer to that structure at the location pointed to by *result*. The structure contains an entry from the group database with a matching *gid* or *name*. Storage referenced by the group structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX} *sysconf* parameter. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

Return Values

The **getgrnam_r** function returns a pointer to a **struct group** with the structure defined in **<grp.h>** with a matching entry if one is found. The **getgrnam_r** function returns a null pointer if either the requested entry was not found, or an error occurred. On error, *errno* will be set to indicate the error.

The return value points to a static area that is overwritten by a subsequent call to the **getgrent**, **getgrgid**, or **getgrnam** subroutine.

If successful, the **getgrnam_r** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **getgrnam_r** function fails if:

ERANGE

Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting **group** structure.

Applications wishing to check for error situations should set *errno* to 0 before calling **getgrnam_r**. If *errno* is set on return, an error occurred.

Related Information

The **getgrent**, **getgrgid**, **getgrnam**, **setgrent**, **endgrent** ("getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine" on page 318) subroutine.

The <grp.h>, imits.h>, and <sys/types.h> header files.

getgroupattr, IDtogroup, nextgroup, or putgroupattr Subroutine

Purpose

Accesses the group information in the user database.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
int getgroupattr (Group, Attribute, Value, Type)
char * Group;
char * Attribute;
void * Value;
int Type;
int putgroupattr (Group, Attribute, Value, Type)
char *Group;
char *Attribute;
void *Value;
int Type;
char *IDtogroup ( GID)
gid_t GID;
char *nextgroup ( Mode, Argument)
int Mode, Argument;
```

Description

Attention: These subroutines and the setpwent and setgrent subroutines should not be used simultaneously. The results can be unpredictable.

These subroutines access group information. Because of their greater granularity and extensibility, you should use them instead of the getgrent, putgrent, getgrnam, getgrgid, setgrent, and endgrent subroutines.

The **getgroupattr** subroutine reads a specified attribute from the group database. If the database is not already open, the subroutine will do an implicit open for reading.

Similarly, the putgroupattr subroutine writes a specified attribute into the group database. If the database is not already open, the subroutine does an implicit open for reading and writing. Data changed by putgroupattr must be explicitly committed by calling the putgroupattr subroutine with a Type parameter specifying the SEC COMMIT value. Until the data is committed, only get subroutine calls within the process will return the written data.

New entries in the user and group databases must first be created by invoking putgroupattr with the SEC NEW type.

The **IDtogroup** subroutine translates a group ID into a group name.

The **nextgroup** subroutine returns the next group in a linear search of the group database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setuserdb** and **enduserdb** subroutines should be used to open and close the user database.

Parameters

Argument Attribute

GID

Group

Mode

Presently unused and must be specified as null.

Specifies which attribute is read. The following possible values are defined in the **usersec.h**

file:

S_ID Group ID. The attribute type is **SEC_INT**.

S_USERS

Members of the group. The attribute type is **SEC_LIST**.

S_ADMS

Administrators of the group. The attribute type is SEC_LIST.

S_ADMIN

Administrative status of a group. Type: SEC_BOOL.

S GRPEXPORT

Specifies if the DCE registry can overwrite the local group information with the DCE group information during a DCE export operation. The attribute type is **SEC_BOOL**.

Additional user-defined attributes may be used and will be stored in the format specified by the *Type* parameter.

Specifies the group ID to be translated into a group name.

Specifies the name of the group for which an attribute is to be read.

Specifies the search mode. Also can be used to delimit the search to one or more user credential databases. Specifying a non-null *Mode* value implicitly rewinds the search. A null mode continues the search sequentially through the database. This parameter specifies one of the following values as a bit mask (defined in the **usersec.h** file):

S LOCAL

The local database of groups are included in the search.

S_SYSTEM

All credentials servers for the system are searched.

000

Туре

Specifies the type of attribute expected. Valid values are defined in the **usersec.h** file and include:

SEC_INT

The format of the attribute is an integer. The buffer returned by the **getgroupattr** subroutine and the buffer supplied by the **putgroupattr** subroutine are defined to contain an integer.

SEC_CHAR

The format of the attribute is a null-terminated character string.

SEC_LIST

The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters.

SEC BOOL

A pointer to an integer (int *) that has been cast to a null pointer.

SEC COMMIT

For the **putgroupattr** subroutine, this value specified by itself indicates that changes to the named group are committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no group is specified, changes to all modified groups are committed to permanent storage.

SEC_DELETE

The corresponding attribute is deleted from the database.

SEC NEW

If using the **putgroupattr** subroutine, updates all the group database files with the new group name.

Value

Specifies the address of a pointer for the **getgroupattr** subroutine. The **getgroupattr** subroutine will return the address of a buffer in the pointer. For the **putgroupattr** subroutine, the *Value* parameter specifies the address of a buffer in which the attribute is stored. See the *Type* parameter for more details.

Security

Files Accessed:

Mode File

rw /etc/group (write access for putgroupattr)

rw /etc/security/group (write access for putgroupattr)

Return Values

The **getgroupattr** and **putgroupattr** subroutines, when successfully completed, return a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

The **IDtogroup** and **nextgroup** subroutines return a character pointer to a buffer containing the requested group name, if successfully completed. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

Note: All of these subroutines return errors from other subroutines.

These subroutines fail if the following is true:

EACCES Access permission is denied for the data request.

The **getgroupattr** and **putgroupattr** subroutines fail if one or more of the following are true:

EINVAL The Value parameter does not point to a valid buffer or to valid data for this type of attribute.

Limited testing is possible and all errors may not be detected.

EINVAL The *Group* parameter is null or contains a pointer to a null string.

EINVAL The Type parameter contains more than one of the SEC_INT, SEC_BOOL, SEC_CHAR,

SEC_LIST, or SEC_COMMIT attributes.

EINVAL The *Type* parameter specifies that an individual attribute is to be committed, and the *Group*

parameter is null.

ENOENT The specified *Group* parameter does not exist or the attribute is not defined for this group.

EPERM Operation is not permitted.

The **IDtogroup** subroutine fails if the following is true:

ENOENT The *GID* parameter could not be translated into a valid group name on the system.

The **nextgroup** subroutine fails if one or more of the following are true:

EINVAL The Mode parameter is not null, and does not specify either S_LOCAL or S_SYSTEM.

ENOENT The *Argument* parameter is not null. **ENOENT** The end of the search was reached.

Related Information

The **getuserattr** ("getuserattr, IDtouser, nextuser, or putuserattr Subroutine" on page 382) subroutine, **getuserpw** ("getuserpw, putuserpw, or putuserpwhist Subroutine" on page 389) subroutine, **setpwdb** subroutine, **setuserdb** subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getgroups Subroutine

Purpose

Gets the supplementary group ID of the current process.

Library

Standard C Library (libc.a)

Syntax

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

int getgroups (NGroups, GIDSet)
int NGroups;
gid_t GIDSet [ ];
```

Description

The **getgroups** subroutine gets the supplementary group ID of the process. The list is stored in the array pointed to by the *GIDSet* parameter. The *NGroups* parameter indicates the number of entries that can be stored in this array. The **getgroups** subroutine never returns more than the number of entries specified by

the NGROUPS_MAX constant. (The NGROUPS_MAX constant is defined in the limits.h file.) If the value in the NGroups parameter is 0, the getgroups subroutine returns the number of groups in the supplementary group.

Parameters

GIDSet Points to the array in which the supplementary group ID of the user's process is stored. **NGroups**

Indicates the number of entries that can be stored in the array pointed to by the GIDSet

parameter.

Return Values

Upon successful completion, the getgroups subroutine returns the number of elements stored into the array pointed to by the GIDSet parameter. If the getgroups subroutine is unsuccessful, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **getgroups** subroutine is unsuccessful if either of the following error codes is true:

EFAULT The NGroups and GIDSet parameters specify an array that is partially or completely outside of

the allocated address space of the process.

EINVAL The NGroups parameter is smaller than the number of groups in the supplementary group.

Related Information

The getgid ("getgid, getegid or gegidx Subroutine" on page 317) subroutine, initgroups ("initgroups Subroutine" on page 456) subroutine, **setgid** subroutine, **setgroups** subroutine.

The groups command, setgroups command.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getgrpaclattr, nextgrpacl, or putgrpaclattr Subroutine

Purpose

Accesses the group screen information in the SMIT ACL database.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
int getgrpaclattr (Group, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value:
int Type;
char *nextgrpacl(void)
int putgrpaclattr (Group, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```

Description

The getgrpaclattr subroutine reads a specified group attribute from the SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the putgrpaclattr subroutine writes a specified attribute into the user SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the putgrpaclattr subroutine must be explicitly committed by calling the putgrpaclattr subroutine with a Type parameter specifying SEC_COMMIT. Until all the data is committed, only the getgrpaclattr subroutine within the process returns written data.

The **nextgrpacl** subroutine returns the next group in a linear search of the group SMIT ACL database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setacldb** and **endacldb** subroutines should be used to open and close the database.

Parameters

Attribute Specifies which attribute is read. The following possible attributes are defined in the usersec.h file:

S SCREENS

String of SMIT screens. The attribute type is **SEC_LIST**.

Specifies the type of attribute expected. Valid types are defined in the usersec.h file and include: Type

SEC LIST

The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series must be an empty (zero character count) string.

For the getgrpaclattr subroutine, the user should supply a pointer to a defined character pointer variable. For the putgrpaclattr subroutine, the user should supply a character pointer.

SEC_COMMIT

For the putgrpaclattr subroutine, this value specified by itself indicates that changes to the named group are to be committed to permanent storage. The Attribute and Value parameters are ignored. If no group is specified, the changes to all modified groups are committed to permanent storage.

SEC DELETE

The corresponding attribute is deleted from the group SMIT ACL database.

SEC NEW

Updates the group SMIT ACL database file with the new group name when using the putgrpaclattr subroutine.

Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the Attribute and Type parameters. See the Type parameter for more details.

Return Values

Value

If successful, the **getgrpaclattr** returns 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Possible return codes are:

EACCES Access permission is denied for the data request.

ENOENT The specified *Group* parameter does not exist or the attribute is not defined for this group.

ENOATTR The specified user attribute does not exist for this group.

EINVAL The Attribute parameter does not contain one of the defined attributes or null. EINVAL The Value parameter does not point to a valid buffer or to valid data for this type of

attribute.

EPERM Operation is not permitted.

Related Information

The **getgrpaclattr**, **nextgrpacl**, or **putgrpaclattr** ("getgrpaclattr, nextgrpacl, or putgrpaclattr Subroutine" on page 326) subroutine, **setacldb**, or **endacldb** subroutine.

getgrset Subroutine

Purpose

Accesses the concurrent group set information in the user database.

Library

Standard C Library (libc.a)

Syntax

char *getgrset (User)
const char * User;

Description

The **getgrset** subroutine returns a pointer to the comma separated list of concurrent group identifiers for the named user.

If the Network Information Service (NIS) is enabled on the system, these subroutines attempt to retrieve the user information from the NIS authentication server.

Parameters

User Specifies the user name.

Return Values

If successful, the **getgrset** subroutine returns a pointer to a list of supplementary groups. This pointer must be freed by the user.

Error Codes

A NULL pointer is returned on error. The value of the errno global variable is undefined on error.

File

/etc/group Contains basic group attributes.

Related Information

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine

Purpose

Manipulates the expiration time of interval timers.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/time.h>
int getinterval ( TimerID, Value)
timer t TimerID;
struct itimerstruc_t *Value;
int incinterval (TimerID, Value, OValue)
timer_t TimerID;
struct itimerstruc t *Value, *OValue;
int absinterval (TimerID, Value, OValue)
timer t TimerID;
struct itimerstruc t *Value, *OValue;
int resabs (TimerID, Resolution, Maximum)
timer_t TimerID;
struct timestruc_t *Resolution, *Maximum;
int resinc (TimerID, Resolution, Maximum)
timer t TimerID;
struct timestruc_t *Resolution, *Maximum;
#include <unistd.h>
unsigned int alarm ( Seconds)
unsigned int Seconds;
useconds_t ualarm (Value, Interval)
useconds t Value, Interval;
int setitimer ( Which, Value, OValue)
int Which;
struct itimerval *Value, *OValue;
int getitimer (Which, Value)
int Which:
struct itimerval *Value;
```

Description

The **getinterval**, **incinterval**, and **absinterval** subroutines manipulate the expiration time of interval timers. These functions use a timer value defined by the struct itimerstruc t structure, which includes the following fields:

```
struct timestruc t it interval; /* timer interval period
struct timestruc t it value;
                               /* timer interval expiration */
```

If the it value field is nonzero, it indicates the time to the next timer expiration. If it value is 0, the per-process timer is disabled. If the it interval member is nonzero, it specifies a value to be used in reloading the it value field when the timer expires. If it interval is 0, the timer is to be disabled after its next expiration (assuming it value is nonzero).

The **getinterval** subroutine returns a value from the **struct itimerstruc** t structure to the *Value* parameter. The it value field of this structure represents the amount of time in the current interval before the timer expires, should one exist for the per-process timer specified in the *TimerID* parameter. The it interval field has the value last set by the incinterval or absinterval subroutine. The fields of the Value parameter are subject to the resolution of the timer.

The incinterval subroutine sets the value of a per-process timer to a given offset from the current timer setting. The absinterval subroutine sets the value of the per-process timer to a given absolute value. If the specified absolute time has already expired, the absinterval subroutine will succeed and the expiration notification will be made. Both subroutines update the interval timer period. Time values smaller than the resolution of the specified timer are rounded up to this resolution. Time values larger than the maximum value of the specified timer are rounded down to the maximum value.

The **resinc** and **resabs** subroutines return the resolution and maximum value of the interval timer contained in the TimerID parameter. The resolution of the interval timer is contained in the Resolution parameter, and the maximum value is contained in the *Maximum* parameter. These values might not be the same as the values returned by the corresponding system timer, the gettimer subroutine. In addition, it is likely that the maximum values returned by the resinc and resabs subroutines will be different.

Note: If a nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of less than 10 milliseconds), the timer request is raised to 10 milliseconds.

The alarm subroutine causes the system to send the calling thread's process a SIGALRM signal after the number of real-time seconds specified by the Seconds parameter have elapsed. Since the signal is sent to the process, in a multi-threaded process another thread than the one that called the alarm subroutine may receive the SIGALRM signal. Processor scheduling delays may prevent the process from handling the signal as soon as it is generated. If the value of the Seconds parameter is 0, a pending alarm request, if any, is canceled. Alarm requests are not stacked. Only one SIGALRM generation can be scheduled in this manner. If the SIGALRM signal has not yet been generated, the call results in rescheduling the time at which the SIGALRM signal is generated. If several threads in a process call the alarm subroutine, only the last call will be effective.

The ualarm subroutine sends a SIGALRM signal to the invoking process in a specified number of seconds. The getitimer subroutine gets the value of an interval timer. The setitimer subroutine sets the value of an interval timer.

Parameters

TimerID Specifies the ID of the interval timer. Value Points to a **struct itimerstruc t** structure. **OValue** Represents the previous time-out period.

Resolution Resolution of the timer.

Indicates the maximum value of the interval timer. Maximum

Seconds Specifies the number of real-time seconds to elapse before the first SIGALRM signal. Specifies the number of microseconds between subsequent periodic SIGALRM signals. If Interval

a nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of

less than 10 milliseconds), the timer request interval is automatically raised to 10

milliseconds.

Which Identifies the type of timer. Valid values are:

ITIMER_REAL

Decrements in real time. A SIGALRM signal occurs when this timer expires.

ITIMER VIRTUAL

Decrements in process virtual time. It runs only during process execution. A SIGVTALRM signal occurs when it expires.

ITIMER PROF

Decrements in process virtual time and when the system runs on behalf of the process. It is designed for use by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER PROF timer expires, the SIGPROF signal occurs. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

Return Values

If these subroutines are successful, a value of 0 is returned. If an error occurs, a value of -1 is returned and the **errno** global variable is set.

The alarm subroutine returns the amount of time (in seconds) remaining before the system is scheduled to generate the SIGALARM signal from the previous call to alarm. It returns a 0 if there was no previous alarm request.

The **ualarm** subroutine returns the number of microseconds previously remaining in the alarm clock.

Error Codes

If the getinterval, incinterval, absinterval, resinc, resabs, setitimer, getitimer, or setitimer subroutine is unsuccessful, a value of -1 is returned and the errno global variable is set to one of the following error codes:

EINVAL Indicates that the TimerID parameter does not correspond to an ID returned by the

gettimerid subroutine, or a value structure specified a nanosecond value less than 0 or

greater than or equal to one thousand million (1,000,000,000).

EIO Indicates that an error occurred while accessing the timer device. **EFAULT** Indicates that a parameter address has referenced invalid memory.

The alarm subroutine is always successful. No return value is reserved to indicate an error for it.

Related Information

The gettimer ("gettimer, settimer, restimer, stime, or time Subroutine" on page 375) subroutine, gettimerid ("gettimerid Subroutine" on page 378) subroutine, sigaction, sigvec, or signal subroutine.

List of Time Data Manipulation Services in AIX 5L Version 5.2 System Management Concepts: Operating System and Devices.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Signal Management in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging *Programs* provides more information about signal management in multi-threaded processes.

getipnodebyaddr Subroutine

Purpose

Address-to-nodename translation.

Library

Standard C Library (libc.a)

(libaixinet)

Syntax

```
#include <sys/socket.h>
#include <netdb.h>
struct hostent *getipnodebyaddr(src, len, af, error num)
const void *src;
size_t len;
int af:
int *error num;
```

Description

The getipnodebyaddr subroutine has the same arguments as the gethostbyaddr subroutine but adds an error number. It is thread-safe.

The **getipnodebyaddr** subroutine is similar in its name guery to the **gethostbyaddr** subroutine except in one case. If af equals AF_INET6 and the IPv6 address is an IPv4-mapped IPv6 address or an IPv4-compatible address, then the first 12 bytes are skipped over and the last 4 bytes are used as an IPv4 address with af equal to AF INET to lookup the name.

If the getipnodebyaddr subroutine is returning success, then the single address that is returned in the hostent structure is a copy of the first argument to the function with the same address family and length that was passed as arguments to this function.

All of the information returned by getipnodebyaddr is dynamically allocated: the hostent structure and the data areas pointed to by the h name, h addr lisy, and h aliases members of the hostent structure. To return this information to the system the function freehostent is called.

Parameters

src	Specifies a node address. It is a pointer to either a 4-byte (IPv4) or 16-byte (IPv6) binary format address.	
af	Specifies the address family which is either AF_INET or AF_INET6.	
len	Specifies the length of the node binary format address.	
error_num	Returns argument to the caller with the appropriate error code.	

Return Values

The **getipnodebyaddr** subroutine returns a pointer to a **hostent** structure on success.

The **getipnodebyaddr** subroutine returns a null pointer if an error occurs. The *error_num* parameter is set to indicate the error.

Error Codes

HOST_NOT_FOUND	The host specified by the <i>name</i> parameter was not found.
TRY_AGAIN	The local server did not receive a response from an authoritative server. Try again later.
NO_RECOVERY	This error code indicates an unrecoverable error.
NO_ADDRESS	The requested <i>name</i> is valid but does not have an Internet address at the name server.

Related Information

The **freehostent** subroutine and **getipnodebyname** subroutine.

getipnodebyname Subroutine

Purpose

Nodename-to-address translation.

Library

Standard C Library (libc.a)

(libaixinet)

Syntax

```
#include <libc.a>
#include <netdb.h>
struct hostent *getipnodebyname(name, af, flags, error num)
const char *name;
int af;
int flags;
int *error num;
```

Description

The commonly used functions gethostbyname and gethostbyname2 are inadequate for many applications. You could not specify the type of addresses desired in gethostbyname. In gethostbyname2, a global option (RES_USE_INET6) is required when IPV6 addresses are used. Also, gethostbyname2 needed more control over the type of addresses required.

The getipnodebyname subroutine gives the caller more control over the types of addresses required and is thread safe. It also does not need a global option like RES USE INET6.

The name argument can be either a node name or a numeric (either a dotted-decimal IPv4 or colon-seperated IPv6) address.

The flags parameter values include AI_DEFAULT, AI_V4MAPPED, AI_ALL and AI_ADDRCONFIG. The special flags value AI_DEFAULT is designed to handle most applications. Its definition is:

```
#define AI DEFAULT (AI V4MAPPED | AI ADDRCONFIG)
```

When porting simple applications to use IPv6, simply replace the call:

```
hp = gethostbyname(name);
```

with

```
hp = getipnodebyname(name, AF INET6, AI DEFAULT, &error num);
```

To modify the behavior of the getipnodebyname subroutine, constant values can be logically-ORed into the *flags* parameter.

A flags value of 0 implies a strict interpretation of the af parameter. If af is AF INET then only IPv4 addresses are searched for and returned. If af is AF INET6 then only IPv6 addresses are searched for and returned.

If the AI_V4MAPPED flag is specified along with an af of AF_INET6, then the caller accepts IPv4-mapped IPv6 addresses. That is, if a query for IPv6 addresses fails, then a query for IPv4 addresses is made and if any are found, then they are returned as IPv4-mapped IPv6 addresses. The AI V4MAPPED flag is only valid with an af of AF INET6.

If the AI_ALL flag is used in conjunction the AI_V4MAPPED flag and af is AF_INET6, then the caller wants all addresses. The addresses returned are IPv6 addresses and/or IPv4-mapped IPv6 addresses. Only if both queries (IPv6 and IPv4) fail does **getipnodebyname** return NULL. Again, the AI ALL flag is only valid with an af of AF INET6.

The AI ADDRCONFIG flag is used to specify that a query for IPv6 addresses should only occur if the node has at least one IPv6 source address configured and a query for IPv4 addresses should only occur if the node has at least one IPv4 source address configured. For example, if the node only has IPv4 addresses configured, af equals AF INET6, and the node name being looked up has both IPv4 and IPv6 addresses, then if only the AI ADDRCONFIG flag is specified, getipnodebyname will return NULL. If the AI_V4MAPPED flag is specified with the AI_ADDRCONFIG flag (AI_DEFAULT), then any IPv4 addresses found will be returned as IPv4-mapped IPv6 addresses.

There are 4 different situations when the name argument is a literal address string:

- 1. name is a dotted-decimal IPv4 address and af is AF_INET. If the query is successful, then h name points to a copy of name, h addrtype is the af argument, h length is 4, h aliases is a NULL pointer, h addr list[0] points to the 4-byte binary address and h addr list[1] is a NULL pointer.
- 2. name is a colon-separated IPv6 address and af is AF_INET6. If the query is successful, then h_name points to a copy of name, h addrtype is the af parameter, h length is 16, h aliases is a NULL pointer, h addr list[0] points to the 16-byte binary address and h addr list[1] is a NULL pointer.
- 3. name is a dotted-decimal IPv4 address and af is AF_INET6. If the AI_V4MAPPED flag is specified and the query is successful, then h name points to an IPv4-mapped IPv6 address string, h addrtype is the af argument, h length is 16, h aliases is a NULL pointer, h addr list[0] points to the 16-byte binary address and h addr list[1] is a NULL pointer.
- 4. name is a colon-separated IPv6 address and af is AF INET. This is an error, getipnodebyname returns a NULL pointer and error num equals HOST NOT FOUND.

Parameters

name	Specifies either a node name or a numeric (either a dotted-decimal IPv4 or colon-separated IPv6) address.
af	Specifies the address family which is either AF_INET or AF_INET6.
flags	Controls the types of addresses searched for and the types of addresses returned.
error_num	Returns argument to the caller with the appropriate error code.

Return Values

The **getipnodebyname** subroutine returns a pointer to a **hostent** structure on success.

The **getipnodebyname** subroutine returns a null pointer if an error occurs. The *error num* parameter is set to indicate the error.

Error Codes

HOST_NOT_FOUND	The host specified by the <i>name</i> parameter was not found.
TRY_AGAIN	The local server did not receive a response from an authoritative server. Try again later.
NO_RECOVERY	The host specified by the <i>name</i> parameter was not found. This error code indicates an unrecoverable error.
NO_ADDRESS	The requested <i>name</i> is valid but does not have an Internet address at the name server.

Related Information

The freehostent subroutine and getipnodebyaddr subroutine.

getlogin Subroutine

Purpose

Gets a user's login name.

Library

Standard C Library (libc.a)

Syntax

include <sys/types.h> include <unistd.h> include <limits.h> char *getlogin (void)

Description

Attention: Do not use the getlogin subroutine in a multithreaded environment. To access the thread-safe version of this subroutines, see the **getlogin_r** ("getlogin_r Subroutine" on page 336) subroutine.

Attention: The getlogin subroutine returns a pointer to an area that may be overwritten by successive calls.

The getlogin subroutine returns a pointer to the login name in the /etc/utmp file. You can use the getlogin subroutine with the getpwnam ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 354) subroutine to locate the correct password file entry when the same user ID is shared by several login names.

If the **getlogin** subroutine cannot find the login name in the /etc/utmp file, it returns the process LOGNAME environment variable. If the getlogin subroutine is called within a process that is not attached to a terminal, it returns the value of the LOGNAME environment variable. If the LOGNAME environment variable does not exist, a null pointer is returned.

Return Values

The return value can point to static data whose content is overwritten by each call. If the login name is not found, the **getlogin** subroutine returns a null pointer.

Error Codes

If the getlogin function is unsuccessful, it returns one or more of the following error codes:

EMFILE Indicates that the **OPEN_MAX** file descriptors are currently open in the calling process. **ENFILE** Indicates that the maximum allowable number of files is currently open in the system.

ENXIO Indicates that the calling process has no controlling terminal.

Files

/etc/utmp Contains a record of users logged into the system.

Related Information

The **getgrent**, **getgrgid**, **getgrnam**, **putgrent**, **setgrent**, or **endgrent** ("getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine" on page 318) subroutine, **getlogin_r** ("getlogin_r Subroutine") subroutine, **getpwent**, **getpwent**, or **endpwent** ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 354) subroutine, **getpwnam** ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 354) subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getlogin_r Subroutine

Purpose

Gets a user's login name.

Library

Thread-Safe C Library (libc_r.a)

Syntax

int getlogin_r (Name, Length)
char * Name;
size_t Length;

Description

The **getlogin_r** subroutine gets a user's login name from the **/etc/utmp** file and places it in the **Name** parameter. Only the number of bytes specified by the **Length** parameter (including the ending null value) are placed in the **Name** parameter.

Applications that call the **getlogin_r** subroutine must allocate memory for the login name before calling the subroutine. The name buffer must be the length of the *Name* parameter plus an ending null value.

If the **getlogin_r** subroutine cannot find the login name in the **utmp** file or the process is not attached to a terminal, it places the **LOGNAME** environment variable in the name buffer. If the **LOGNAME** environment variable does not exist, the *Name* parameter is set to null and the **getlogin_r** subroutine returns a -1.

Parameters

Name Specifies a buffer for the login name. This buffer should be the length of the *Length* parameter plus an ending null value.

Length

Specifies the total length in bytes of the Name parameter. No more bytes than the number specified by the Length parameter are placed in the Name parameter, including the ending null value.

Return Values

- Indicates that the subroutine was successful.
- Indicates that the subroutine was not successful.

Error Codes

If the **getlogin_r** subroutine does not succeed, it returns one of the following error codes:

EMFILE Indicates that the **OPEN_MAX** file descriptors are currently open in the calling process. **ENFILE** Indicates that the maximum allowable number of files are currently open in the system.

ENXIO Indicates that the calling process has no controlling terminal.

File

/etc/utmp Contains a record of users logged into the system.

Related Information

The getgrent_r, getgrgid_r, getgrnam_r, setgrent_r, or endgrent_r ("getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine" on page 318) subroutine, getlogin ("getlogin Subroutine" on page 335) subroutine, getpwent_r, getpwnam_r, putpwent_r, getpwuid_r, setpwent_r, or endpwent_r ("getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine" on page 354) subroutine.

List of Security and Auditing Subroutines, List of Multithread Subroutines, and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getopt Subroutine

Purpose

Returns the next flag letter specified on the command line.

Library

Standard C Library (libc.a)

Syntax

```
#include <unistd.h>
int getopt (ArgumentC, ArgumentV, OptionString)
int ArgumentC;
char *const ArgumentV [ ];
const char *OptionString;
extern int optind;
extern int optopt;
extern int opterr;
extern char * optarg;
```

Description

The optind parameter indexes the next element of the ArgumentV parameter to be processed. It is initialized to 1 and the getopt subroutine updates it after calling each element of the ArgumentV parameter.

The **getopt** subroutine returns the next flag letter in the *ArgumentV* parameter list that matches a letter in the OptionString parameter. If the flag takes an argument, the getopt subroutine sets the optarg parameter to point to the argument as follows:

- If the flag was the last letter in the string pointed to by an element of the ArgumentV parameter, the optarg parameter contains the next element of the ArgumentV parameter and the optind parameter is incremented by 2. If the resulting value of the optind parameter is not less than the ArgumentC parameter, this indicates a missing flag argument, and the **getopt** subroutine returns an error message.
- · Otherwise, the optarg parameter points to the string following the flag letter in that element of the ArgumentV parameter and the optind parameter is incremented by 1.

Parameters

ArgumentC Specifies the number of parameters passed to the routine. ArgumentV Specifies the list of parameters passed to the routine.

Specifies a string of recognized flag letters. If a letter is followed by a: (colon), the flag is **OptionString**

expected to take a parameter that may or may not be separated from it by white space.

optind Specifies the next element of the *ArgumentV* array to be processed. optopt Specifies any erroneous character in the *OptionString* parameter. opterr Indicates that an error has occurred when set to a value other than 0.

Points to the next option flag argument. optarg

Return Values

The getopt subroutine returns the next flag letter specified on the command line. A value of -1 is returned when all command line flags have been parsed. When the value of the ArgumentV [optind] parameter is null, *ArgumentV [optind] is not the - (minus) character, or ArgumentV [optind] points to the "-" (minus) string, the **getopt** subroutine returns a value of -1 without changing the value. If ArgumentV [optind] points to the "--" (double minus) string, the getopt subroutine returns a value of -1 after incrementing the value of the optind parameter.

Error Codes

If the **getopt** subroutine encounters an option character that is not specified by the *OptionString* parameter, a ? (question mark) character is returned. If it detects a missing option argument and the first character of OptionString is a: (colon), then a: (colon) character is returned. If this subroutine detects a missing option argument and the first character of OptionString is not a colon, it returns a ? (question mark). In either case, the **getopt** subroutine sets the *optopt* parameter to the option character that caused the error. If the application has not set the opterr parameter to 0 and the first character of OptionString is not a: (colon), the **getopt** subroutine also prints a diagnostic message to standard error.

Examples

The following code fragment processes the flags for a command that can take the mutually exclusive flags **a** and **b**, and the flags **f** and **o**, both of which require parameters.

```
#include <unistd.h>
                       /*Needed for access subroutine constants*/
main (argc, argv)
int argc;
char **argv;
   int c;
   extern int optind;
   extern char *optarg;
```

```
while ((c = getopt(argc, argv, "abf:o:")) != EOF)
{
   switch (c)
      case 'a':
         if (bflg)
            errflg++;
         else
            aflg++;
         break;
      case 'b':
         if (aflg)
            errflg++;
            bflg++;
         break;
      case 'f':
         ifile = optarg;
         break;
      case 'o':
         ofile = optarg;
         break;
      case '?':
        errflg++;
   } /* case */
   if (errflg)
      fprintf(stderr, "usage: . . . ");
      exit(2);
} /* while */
for ( ; optind < argc; optind++)</pre>
   if (access(argv[optind], R_OK))
} /* for */
/* main */
```

Related Information

The getopt command.

List of Executable Program Creation Subroutines, Subroutines Overview, and List of Multithread Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getpagesize Subroutine

Purpose

Gets the system page size.

Library

Standard C Library (libc.a)

Syntax

#include <unistd.h>

int getpagesize()

Description

The **getpagesize** subroutine returns the number of bytes in a page. Page granularity is the granularity for many of the memory management calls.

The page size is determined by the system and may not be the same as the underlying hardware page size.

Related Information

The **brk** or **sbrk** ("brk or sbrk Subroutine" on page 100) subroutine.

The pagesize command.

Program Address Space Overview and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getpass Subroutine

Purpose

Reads a password.

Library

Standard C Library (libc.a)

Syntax

#include <stdlib.h>
char *getpass (Prompt)
char *Prompt;

Description

Attention: The characters are returned in a static data area. Subsequent calls to this subroutine overwrite the static data area.

The **getpass** subroutine does the following:

- · Opens the controlling terminal of the current process.
- Writes the characters specified by the *Prompt* parameter to that device.
- Reads from that device the number of characters up to the value of the PASS_MAX constant until a new-line or end-of-file (EOF) character is detected.
- · Restores the terminal state and closes the controlling terminal.

During the read operation, character echoing is disabled.

The **getpass** subroutine is not safe in a multithreaded environment. To use the **getpass** subroutine in a threaded application, the application must keep the integrity of each thread.

Parameters

Prompt Specifies a prompt to display on the terminal.

Return Values

If this subroutine is successful, it returns a pointer to the string. If an error occurs, the subroutine returns a null pointer and sets the **errno** global variable to indicate the error.

Error Codes

If the **getpass** subroutine is unsuccessful, it returns one or more of the following error codes:

EINTR Indicates that an interrupt occurred while the getpass subroutine was reading the terminal device. If a

SIGINT or SIGQUIT signal is received, the getpass subroutine terminates input and sends the signal to

the calling process.

ENXIO Indicates that the process does not have a controlling terminal.

Note: Any subroutines called by the **getpass** subroutine may set other error codes.

Related Information

The getuserpw ("getuserpw, putuserpw, or putuserpwhist Subroutine" on page 389) subroutine, newpass ("newpass Subroutine" on page 732) subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getpcred Subroutine

Purpose

Reads the current process credentials.

Library

Security Library (libc.a)

Syntax

#include <usersec.h>

char **getpcred (Which) int Which;

Description

The getpcred subroutine reads the specified process security credentials and returns a pointer to a NULL terminated array of pointers in allocated memory. Each pointer in the array points to a string containing an attribute/value pair in allocated memory. It's the responsibility of the caller to free each individual string as well as the array of pointers.

Parameters

Which

Specifies which credentials are read. This parameter is a bit mask and can contain one or more of the following values, as defined in the usersec.h file:

CRED RUID

Real user name

CRED_LUID

Login user name

CRED_RGID

Real group name

CRED GROUPS

Supplementary group ID

CRED AUDIT

Audit class of the current process

Note: A process must have root user authority to retrieve this credential. Otherwise, the getpcred subroutine returns a null pointer and the errno global variable is set to EPERM.

CRED RLIMITS

BSD resource limits

Note: Use the getrlimit ("getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit Subroutine" on page 356) subroutine to control resource consumption.

CRED_UMASK

The umask.

If the Which parameter is null, all credentials are returned.

Return Values

When successful, the **getpcred** subroutine returns a pointer to a NULL terminated array of string pointers containing the requested values. If the getpcred subroutine is unsuccessful, a NULL pointer is returned and the errno global variable is set to indicate the error.

Error Codes

The **getpcred** subroutine fails if either of the following are true:

EINVAL The Which parameter contains invalid credentials requests.

EPERM The process does not have the proper authority to retrieve the requested credentials.

Other errors can also be set by any subroutines invoked by the getpcred subroutine.

Related Information

The ckuseracct ("ckuseracct Subroutine" on page 135) subroutine, ckuserID ("ckuserID Subroutine" on page 137) subroutine, getpenv ("getpenv Subroutine" on page 343) subroutine, setpenv subroutine, **setpcred** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getpenv Subroutine

Purpose

Reads the current process environment.

Library

Security Library (libc.a)

Syntax

#include <usersec.h>

char **getpenv (Which) int Which;

Description

The getpenv subroutine reads the specified environment variables and returns them in a character buffer.

Parameters

Which

Specifies which environment variables are to be returned. This parameter is a bit mask and may contain one or more of the following values, as defined in the usersec.h file:

PENV USR

The normal user-state environment. Typically, the shell variables are contained here.

PENV SYS

The system-state environment. This data is located in system space and protected from unauthorized access.

All variables are returned by setting the Which parameter to logically OR the PENV_USER and PENV_SYSTEM values.

The variables are returned in a null-terminated array of character pointers in the form var=val. The user-state environment variables are prefaced by the string **USRENVIRON**:, and the system-state variables are prefaced with SYSENVIRON:. If a user-state environment is requested, the current directory is always returned in the PWD variable. If this variable is not present in the existing environment, the getpenv subroutine adds it to the returned string.

Return Values

Upon successful return, the getpenv subroutine returns the environment values. If the getpenv subroutine fails, a null value is returned and the errno global variable is set to indicate the error.

Note: This subroutine can partially succeed, returning only the values that the process permits it to read.

Error Codes

The **getpenv** subroutine fails if one or more of the following are true:

EINVAL The Which parameter contains values other than PENV_USR or PENV_SYS.

Other errors can also be set by subroutines invoked by the **getpenv** subroutine.

Related Information

The **ckuseracct** ("ckuseracct Subroutine" on page 135) subroutine, **ckuserID** ("ckuserID Subroutine" on page 137) subroutine, **getpcred** ("getpcred Subroutine" on page 341) subroutine, **setpenv** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getpgid Subroutine

Purpose

Returns the process group ID of the calling process.

Library

Standard C Library (libc.a)

Syntax

#include <unistd.h>
pid_t getpgid (Pid)
(pid_ Pid)

Description

The **getpgid** subroutine returns the process group ID of the process whose process ID is equal to that specified by the *Pid* parameter. If the value of the *Pid* parameter is equal to **(pid_t)0**, the **getpgid** subroutine returns the process group ID of the calling process.

Parameter

Pid The process ID of the process to return the process group ID for.

Return Values

id The process group ID of the requested process-1 Not successful and errno set to one of the following.

Error Code

ESRCH There is no process with a process ID equal to *Pid*.

EPERM The process whose process ID is equal to *Pid* is not in the same session as the calling

process.

EINVAL The value of the *Pid* argument is invalid.

Related Information

The **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutine, **fork** ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, **getpid** ("getpid, getpgrp, or getppid Subroutine" on page 345) subroutine, **getsid** ("getsid Subroutine" on page 368) subroutine, **setpgid** subroutine, **setsid** subroutine.

getpid, getpgrp, or getppid Subroutine

Purpose

Returns the process ID, process group ID, and parent process ID.

Syntax

```
#include <unistd.h>
pid_t getpid (void)
pid t getpgrp (void)
pid_t getppid (void)
```

Description

The **getpid** subroutine returns the process ID of the calling process.

The **getpgrp** subroutine returns the process group ID of the calling process.

The **getppid** subroutine returns the process ID of the calling process' parent process.

Related Information

The **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutines, fork ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, setpgid subroutine, setpgrp subroutine, sigaction, sigvec, or signal subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getportattr or putportattr Subroutine

Purpose

Accesses the port information in the port database.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
int getportattr (Port, Attribute, Value, Type)
char * Port;
char * Attribute;
void * Value;
int Type;
int putportattr (Port, Attribute, Value, Type)
char *Port;
char *Attribute;
void *Value;
int Type;
```

Description

The getportattr or putportattr subroutine accesses port information. The getportattr subroutine reads a specified attribute from the port database. If the database is not already open, the getportattr subroutine implicitly opens the database for reading. The **putportattr** subroutine writes a specified attribute into the port database. If the database is not already open, the **putportattr** subroutine implicitly opens the database for reading and writing. The data changed by the **putportattr** subroutine must be explicitly committed by calling the **putportattr** subroutine with a *Type* parameter equal to the **SEC_COMMIT** value. Until all the data is committed, only these subroutines within the process return the written data.

Values returned by these subroutines are in dynamically allocated buffers. You do not need to move the values prior to the next call.

Use the **setuserdb** or **enduserdb** subroutine to open and close the port database.

Parameters

Port

Specifies the name of the port for which an attribute is read.

Attribute

Specifies the name of the attribute read. This attribute can be one of the following values defined in the usersec.h file:

S HERALD

Defines the initial message printed when the **getty** or **login** command prompts for a login name. This value is of the type **SEC_CHAR**.

S SAKENABLED

Indicates whether or not trusted path processing is allowed on this port. This value is of the type **SEC BOOL**.

S SYNONYM

Defines the set of ports that are **synonym** attributes for the given port. This value is of the type **SEC_LIST**.

S_LOGTIMES

Defines when the user can access the port. This value is of the type SEC_LIST.

S LOGDISABLE

Defines the number of unsuccessful login attempts that result in the system locking the port. This value is of the type **SEC_INT**.

S_LOGINTERVAL

Defines the time interval in seconds within which **S_LOGDISABLE** number of unsuccessful login attempts must occur before the system locks the port. This value is of the type **SEC_INT**.

S LOGREENABLE

Defines the time interval in minutes after which a system-locked port is unlocked. This value is of the type **SEC_INT**.

S_LOGDELAY

Defines the delay factor in seconds between unsuccessful login attempts. This value is of the type **SEC_INT**.

S_LOCKTIME

Defines the time in seconds since the epoch (zero time, January 1, 1970) that the port was locked. This value is of the type **SEC_INT**.

S_ULOGTIMES

Lists the times in seconds since the epoch (midnight, January 1, 1970) when unsuccessful login attempts occurred. This value is of the type **SEC_LIST**.

S USERNAMEECHO

Indicates whether user name input echo and user name masking is enabled for the port. This value is of the type **SEC_BOOL**.

S_PWDPROMPT

Defines the password prompt message printed when requesting password input. This value is of the type **SEC_CHAR**.

Value

Specifies the address of a buffer in which the attribute is stored with **putportattr** or is to be read **getportattr**.

Type

Specifies the type of attribute expected. The following types are valid and defined in the **usersec.h** file:

SEC INT

Indicates the format of the attribute is an integer. The buffer returned by the **getportattr** subroutine and the buffer supplied by the **putportattr** subroutine are defined to contain an integer.

SEC_CHAR

Indicates the format of the attribute is a null-terminated character string.

SEC LIST

Indicates the format of the attribute is a list of null-terminated character strings. The list itself is null terminated.

SEC BOOL

An integer with a value of either 0 or 1, or a pointer to a character pointing to one of the following strings:

- True
- Yes
- Always
- False
- No
- Never

SEC_COMMIT

Indicates that changes to the specified port are committed to permanent storage if specified alone for the **putportattr** subroutine. The *Attribute* and *Value* parameters are ignored. If no port is specified, changes to all modified ports are committed.

SEC DELETE

Deletes the corresponding attribute from the database.

SEC_NEW

Updates all of the port database files with the new port name when using the **putportattr** subroutine.

Security

Access Control: The calling process must have access to the port information in the port database.

File Accessed:

rw /etc/security/login.cfg
rw /etc/security/portlog

Return Values

The **getportattr** and **putportattr** subroutines return a value of 0 if completed successfully. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

These subroutines are unsuccessful if the following values are true:

EACCES Indicates that access permission is denied for the data requested.

ENOENT Indicates that the *Port* parameter does not exist or the attribute is not defined for the

specified port.

ENOATTR Indicates that the specified port attribute does not exist for the specified port.

EINVAL Indicates that the Attribute parameter does not contain one of the defined attributes or is a

null value.

EINVAL Indicates that the Value parameter does not point to a valid buffer or to valid data for this

type of attribute.

EPERM Operation is not permitted.

Related Information

The **setuserdb** or **enduserdb** subroutine.

List of Security and Auditing Services in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getpri Subroutine

Purpose

Returns the scheduling priority of a process.

Library

Standard C Library (libc.a)

Syntax

int getpri (ProcessID)
pid_t ProcessID;

Description

The getpri subroutine returns the scheduling priority of a process.

Parameters

ProcessID Specifies the process ID. If this value is 0, the current process scheduling priority is returned.

Return Values

Upon successful completion, the **getpri** subroutine returns the scheduling priority of a thread in the process. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The getpri subroutine is unsuccessful if one of the following is true:

EPERM A process was located, but its effective and real user ID did not match those of the process

executing the getpri subroutine, and the calling process did not have root user authority.

ESRCH No process can be found corresponding to that specified by the *ProcessID* parameter.

Related Information

The **setpri** subroutine.

Performance-Related Subroutines in AIX 5L Version 5.2 Performance Management Guide.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getpriority, setpriority, or nice Subroutine

Purpose

Gets or sets the nice value.

Libraries

```
getpriority, setpriority: Standard C Library (libc.a)
nice: Standard C Library (libc.a)
Berkeley Compatibility Library (libbsd.a)
```

Syntax

```
#include <sys/resource.h>
int getpriority( Which, Who)
int Which;
int Who;
int setpriority(Which, Who, Priority)
int Which;
int Who;
int Priority;
#include <unistd.h>
int nice( Increment)
int Increment;
```

Description

The nice value of the process, process group, or user, as indicated by the Which and Who parameters is obtained with the getpriority subroutine and set with the setpriority subroutine.

The getpriority subroutine returns the highest priority nice value (lowest numerical value) pertaining to any of the specified processes. The setpriority subroutine sets the nice values of all of the specified processes to the specified value. If the specified value is less than -20, a value of -20 is used; if it is greater than 20, a value of 20 is used. Only processes that have root user authority can lower nice values.

The **nice** subroutine increments the nice value by the value of the *Increment* parameter.

Note: Nice values are only used for the scheduling policy SCHED_OTHER, where they are combined with a calculation of recent cpu usage to determine the priority value.

To provide upward compatibility with older programs, the **nice** interface, originally found in AT&T System V, is supported.

Note: Process priorities in AT&T System V are defined in the range of 0 to 39, rather than -20 to 20 as in BSD, and the **nice** library routine is supported by both. Accordingly, two versions of the **nice** are supported by AIX Version 3. The default version behaves like the AT&T System V version, with the Increment parameter treated as the modifier of a value in the range of 0 to 39 (0 corresponds to -20, 39 corresponds to 9, and priority 20 is not reachable with this interface).

If the behavior of the BSD version is desired, compile with the Berkeley Compatibility Library (libbsd.a). The *Increment* parameter is treated as the modifier of a value in the range -20 to 20.

Parameters

Which Specifies one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.

Who Interpreted relative to the Which parameter (a process identifier, process group identifier, and

a user ID, respectively). A zero value for the Who parameter denotes the current process,

process group, or user.

Priority Specifies a value in the range -20 to 20. Negative nice values cause more favorable

scheduling.

Specifies a value that is added to the current process nice value. Negative values can be Increment

specified, although values exceeding either the high or low limit are truncated.

Return Values

On successful completion, the getpriority subroutine returns an integer in the range -20 to 20. A return value of -1 can also indicate an error, and in this case the errno global variable is set.

On successful completion, the **setpriority** subroutine returns 0. Otherwise, -1 is returned and the global variable **errno** is set to indicate the error.

On successful completion, the nice subroutine returns the new nice value minus {NZERO}. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Note: A value of -1 can also be returned. In that case, the calling process should also check the errno global variable.

Error Codes

The getpriority and setpriority subroutines are unsuccessful if one of the following is true:

ESRCH No process was located using the Which and Who parameter values specified.

EINVAL The Which parameter was not recognized.

In addition to the errors indicated above, the **setpriority** subroutine is unsuccessful if one of the following is true:

EPERM A process was located, but neither the effective nor real user ID of the caller of the process

executing the setpriority subroutine has root user authority.

The call to **setpriority** would have changed the priority of a process to a value lower than **EACCESS**

its current value, and the effective user ID of the process executing the call did not have

root user authority.

The **nice** subroutine is unsuccessful if the following is true:

EPERM The Increment parameter is negative or greater than 2 * {NZERO} and the calling process

does not have appropriate privileges.

Related Information

The **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutines.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getprocs Subroutine

Purpose

Gets process table entries.

Library

Standard C library (libc.a)

Syntax

```
#include cinfo.h>
#include <sys/types.h>
int
getprocs ( ProcessBuffer, ProcessSize, FileBuffer, FileSize, IndexPointer, Count)
struct procsinfo *ProcessBuffer;
or struct procsinfo64 *ProcessBuffer;
int ProcessSize;
struct fdsinfo *FileBuffer;
int FileSize;
pid t *IndexPointer;
int Count;
getprocs64 ( ProcessBuffer, ProcessSize, FileBuffer, FileSize, IndexPointer, Count)
struct procentry64 *ProcessBuffer;
int ProcessSize;
struct fdsinfo64 *FileBuffer;
int FileSize;
pid t *IndexPointer;
int Count;
```

Description

The **getprocs** subroutine returns information about processes, including process table information defined by the **procsinfo** structure, and information about the per-process file descriptors defined by the **fdsinfo** structure.

The **getprocs** subroutine retrieves up to *Count* process table entries, starting with the process table entry corresponding to the process identifier indicated by IndexPointer, and places them in the array of procsinfo structures indicated by the *ProcessBuffer* parameter. File descriptor information corresponding to the retrieved processes are stored in the array of fdsinfo structures indicated by the FileBuffer parameter.

On return, the process identifier referenced by *IndexPointer* is updated to indicate the next process table entry to be retrieved. The **getprocs** subroutine returns the number of process table entries retrieved.

The **getprocs** subroutine is normally called repeatedly in a loop, starting with a process identifier of zero, and looping until the return value is less than Count, indicating that there are no more entries to retrieve.

Note: The process table may change while the getprocs subroutine is accessing it. Returned entries will always be consistent, but since processes can be created or destroyed while the getprocs subroutine is running, there is no guarantee that retrieved entries will still exist, or that all existing processes have been retrieved.

When used in 32-bit mode, limits larger than can be represented in 32 bits are truncated to RLIM_INFINITY. Large rusage and other values are truncated to INT_MAX. Alternatively, the struct procsinfo64 and sizeof (struct procsinfo64) can be used by 32-bit getprocs to return full 64-bit process information. Note that the procsinfo structure not only increases certain procsinfo fields from 32 to 64 bits, but that it contains additional information not present in procsinfo. The struct procsinfo64 contains the same data as **struct procsinfo** when compiled in a 64-bit program.

In AIX 5.1 and later, 64-bit applications are required to use **getprocs64()** and **procentry64**. Note that struct procentry64 contains the same information as struct procsinfo64, with the addition of support for the 64-bit time t and dev t, and the 256-bit sigset t. The procentry64 structure also contains a new version of struct ucred (struct ucred ext) and a new, expanded struct rusage (struct trusage64) as described in <svs/cred.h> and <svs/resource.h> respectively. Application developers are also encouraged to use getprocs64() in 32-bit applications to obtain 64-bit process information as this interface provides the new, larger types. The getprocs() interface will still be supported for 32-bit applications using struct procsinfo or struct procsinfo64 but will not be available to 64-bit applications.

Parameters

ProcessBuffer

Specifies the starting address of an array of procsinfo, procsinfo64, or procentry64 structures to be filled in with process table entries. If a value of **NULL** is passed for this parameter, the getprocs subroutine scans the process table and sets return values as normal, but no process entries are retrieved.

Note: The *ProcessBuffer* parameter of **getprocs** subroutine contains two struct rusage fields named pi ru and pi cru. Each of these fields contains two struct timeval fields named ru utime and ru stime. The tv usec field in both of the struct timeval contain nanoseconds instead of microseconds. These values cone from the struct user fields named U_ru and U_cru.

ProcessSize

Specifies the size of a single procsinfo, procsinfo4, or procentry64 structure.

FileBuffer

Specifies the starting address of an array of fdsinfo, or fdsinfo64 structures to be filled in with per-process file descriptor information. If a value of NULL is passed for this parameter, the getprocs subroutine scans the process table and sets return values as normal, but no file descriptor entries are retrieved.

FileSize

Specifies the size of a single **fdsinfo**, or **fdsinfo64** structure.

IndexPointer

Specifies the address of a process identifier which indicates the required process table entry. A process identifier of zero selects the first entry in the table. The process identifier is updated to indicate the next entry to be retrieved.

Note: The IndexPointer does not have to correspond to an existing process, and may in fact correspond to a different process than the one you expect. There is no guarantee that the process slot pointed to by IndexPointer will contain the same process between successive calls to getprocs() or getprocs64().

Count Specifies the number of process table entries requested.

Return Values

If successful, the getprocs subroutine returns the number of process table entries retrieved; if this is less than the number requested, the end of the process table has been reached. Otherwise, a value of -1 is returned, and the errno global variable is set to indicate the error.

Error Codes

The **getprocs** subroutine does not succeed if the following are true:

EINVAL The ProcessSize or FileSize parameters are invalid, or the IndexPointer parameter does not

point to a valid process identifier, or the Count parameter is not greater than zero.

EFAULT The copy operation to one of the buffers was not successful.

Related Information

The getpid ("getpid, getpgrp, or getppid Subroutine" on page 345), getpgrp ("getpid, getpgrp, or getppid Subroutine" on page 345), or getppid ("getpid, getpgrp, or getppid Subroutine" on page 345) subroutines, the **getthrds** ("getthrds Subroutine" on page 372) subroutine

The **ps** command.

getpw Subroutine

Purpose

Retrieves a user's /etc/passwd file entry.

Library

Standard C Library (libc.a)

Syntax

int getpw (UserID, Buffer)

uid t UserID char *Buffer

Description

The getpw subroutine opens the /etc/passwd file and returns, in the Buffer parameter, the /etc/passwd file entry of the user specified by the *UserID* parameter.

Parameters

Buffer Specifies a character buffer large enough to hold any /etc/passwd entry.

UserID Specifies the ID of the user for which the entry is desired.

Return Values

The **getpw** subroutine returns:

0 Successful completion

-1 Not successful.

getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine

Purpose

Accesses the basic user information in the user database.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwent ( )
struct passwd *getpwuid ( UserID)
uid t UserID;
struct passwd *getpwnam ( Name)
char *Name;
int putpwent ( Password, File)
struct passwd *Password;
FILE *File;
void setpwent ( )
void endpwent ( )
```

Description

Attention: All information generated by the getpwent, getpwnam, and getpwuid subroutines is stored in a static area. Subsequent calls to these subroutines overwrite this static area. To save the information in the static area, applications should copy it.

These subroutines access the basic user attributes.

The **setpwent** subroutine opens the user database if it is not already open. Then, this subroutine sets the cursor to point to the first user entry in the database. The **endpwent** subroutine closes the user database.

The getpwent, getpwnam, and getpwuid subroutines return information about a user. These subroutines do the following:

getpwent Returns the next user entry in the sequential search.

getpwnam Returns the first user entry in the database whose name matches the *Name* parameter. getpwuid Returns the first user entry in the database whose ID matches the *UserID* parameter.

The putpwent subroutine writes a password entry into a file in the colon-separated format of the /etc/passwd file.

The user Structure

The getpwent, getpwnam, and getpwuid subroutines return a user structure. This structure The user structure is defined in the pwd.h file and has the following fields:

pw name Contains the name of the user name. Contains the user's encrypted password. pw passwd

> Note: If the password is not stored in the /etc/passwd file and the invoker does not have access to the shadow file that contains passwords, this field contains an undecryptable string,

usually an * (asterisk).

pw uid Contains the user's ID.

pw gid Identifies the user's principal group ID. Contains general user information. pw gecos pw dir Identifies the user's home directory. pw shell Identifies the user's login shell.

Note: If Network Information Services (NIS) is enabled on the system, these subroutines attempt to retrieve the information from the NIS authentication server before attempting to retrieve the information locally.

Parameters

File Points to an open file whose format is similar to the /etc/passwd file format.

Name Specifies the user name.

Password Points to a password structure. This structure contains user attributes.

UserID Specifies the user ID.

Security

Files Accessed:

Mode

/etc/passwd (write access for the putpwent subroutine only) rw

/etc/security/passwd (if the password is desired)

Return Values

The getpwent, getpwnam, and getpwuid subroutines return a pointer to a valid password structure if successful. Otherwise, a null pointer is returned.

The getpwent subroutine will return a null pointer and an errno value of ENOATTR when it detects a corrupt entry. To get subsequent entries following the corrupt entry, call the getpwent subroutine again.

Files

/etc/passwd Contains user IDs and their passwords

Related Information

The getgrent ("getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine" on page 318) subroutine, getgroupattr ("getgroupattr, IDtogroup, nextgroup, or putgroupattr Subroutine" on page 322) subroutine, getuserattr ("getuserattr, IDtouser, nextuser, or putuserattr Subroutine" on page 382) subroutine, getuserpw, putuserpw, or putuserpwhist ("getuserpw, putuserpw, or putuserpwhist Subroutine" on page 389) subroutine, setuserdb subroutine.

List of Security and Auditing Subroutines, Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit Subroutine

Purpose

Controls maximum system resource consumption.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/time.h>
#include <sys/resource.h>
int setrlimit( Resource1, RLP)
int Resource1;
struct rlimit *RLP:
int setrlimit64 ( Resource1, RLP)
int Resource1;
struct rlimit64 *RLP;
int getrlimit ( Resource1, RLP)
int Resource1;
struct rlimit *RLP;
int getrlimit64 ( Resource1, RLP)
int Resource1;
struct rlimit64 *RLP;
#include <sys/vlimit.h>
vlimit ( Resource2, Value)
int Resource2, Value;
```

Description

The getrlimit subroutine returns the values of limits on system resources used by the current process and its children processes. The setrlimit subroutine sets these limits. The vlimit subroutine is also supported, but the getrlimit subroutine replaces it.

A resource limit is specified as either a soft (current) or hard limit. A calling process can raise or lower its own soft limits, but it cannot raise its soft limits above its hard limits. A calling process must have root user authority to raise a hard limit.

Note: The initial values returned by the getrlimit subroutine are the ulimit values in effect when the process was started. For maxdata programs the initial soft limit for data is set to the lower of data ulimit value or a value corresponding to the number of data segments reserved for data segments.

The **rlimit** structure specifies the hard and soft limits for a resource, as defined in the **sys/resource.h** file. The **RLIM INFINITY** value defines an infinite value for a limit.

When compiled in 32-bit mode, RLIM INFINITY is a 32-bit value; when compiled in 64-bit mode, it is a 64-bit value. 32-bit routines should use RLIM64 INFINITY when setting 64-bit limits with the setrlimit64 routine, and recognize this value when returned by getrlimit64.

This information is stored as per-process information. This subroutine must be executed directly by the shell if it is to affect all future processes created by the shell.

Note: Raising the data limit does not raise the program break value. Use the brk/sbrk subroutines to raise the break value. If the proper memory segments are not initialized at program load time, raising your memory limit will not allow access to this memory. Use the -bmaxdata flag of the Id command to set up these segments at load time.

When compiled in 32-bit mode, the struct rlimit values may be returned as RLIM_SAVED_MAX or RLIM_SAVED_CUR when the actual resource limit is too large to represent as a 32-bit rlim_t.

These values can be used by library routines which set their own rlimits to save off potentially 64-bit rlimit values (and prevent them from being truncated by the 32-bit struct rlimit). Unless the library routine intends to permanently change the rlimits, the RLIM SAVED MAX and RLIM SAVED CUR values can be used to restore the 64-bit rlimits.

Application limits may be further constrained by available memory or implementation defined constants such as **OPEN MAX** (maximum available open files).

Parameters

Resource1

Can be one of the following values:

RLIMIT AS

The maximum size of a process' total available memory, in bytes. This limit is not enforced.

RLIMIT_CORE

The largest size, in bytes, of a core file that can be created. This limit is enforced by the kernel. If the value of the RLIMIT_FSIZE limit is less than the value of the RLIMIT_CORE limit, the system uses the RLIMIT_FSIZE limit value as the soft limit.

RLIMIT_CPU

The maximum amount of central processing unit (CPU) time, in seconds, to be used by each process. If a process exceeds its soft CPU limit, the kernel will send a SIGXCPU signal to the process.

RLIMIT DATA

The maximum size, in bytes, of the data region for a process. This limit defines how far a program can extend its break value with the **sbrk** subroutine. This limit is enforced by the kernel.

RLIMIT FSIZE

The largest size, in bytes, of any single file that can be created. When a process attempts to write, truncate, or clear beyond its soft RLIMIT FSIZE limit, the operation will fail with errno set to EFBIG. If the environment variable XPG SUS ENV=ON is set in the user's environment before the process is executed, then the SIGXFSZ signal is also generated.

RLIMIT NOFILE

This is a number one greater than the maximum value that the system may assign to a newly-created descriptor.

RLIMIT_STACK

The maximum size, in bytes, of the stack region for a process. This limit defines how far a program stack region can be extended. Stack extension is performed automatically by the system. This limit is enforced by the kernel. When the stack limit is reached, the process receives a SIGSEGV signal. If this signal is not caught by a handler using the signal stack, the signal ends the process.

RLIMIT RSS

The maximum size, in bytes, to which the resident set size of a process can grow. This limit is not enforced by the kernel. A process may exceed its soft limit size without being ended.

Points to the rlimit or rlimit64 structure, which contains the soft (current) and hard limits. For the getrlimit subroutine, the requested limits are returned in this structure. For the setrlimit

subroutine, the desired new limits are specified here.

The flags for this parameter are defined in the sys/vlimit.h, and are mapped to

corresponding flags for the setrlimit subroutine.

Value Specifies an integer used as a soft-limit parameter to the vlimit subroutine.

Return Values

On successful completion, a return value of 0 is returned, changing or returning the resource limit. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error. If the current limit specified is beyond the hard limit, the setrlimit subroutine sets the limit to to max limit and returns successfully.

Error Codes

The getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit subroutine is unsuccessful if one of the following is true:

RLP

Resource2

The address specified for the RLP parameter is not valid. **EFAULT**

EINVAL The Resource1 parameter is not a valid resource, or the limit specified in the RLP parameter

is invalid.

EPERM The limit specified to the setrlimit subroutine would have raised the maximum limit value.

and the caller does not have root user authority.

Related Information

The sigaction, sigvec, or signal subroutines, sigstack subroutine, ulimit subroutine.

getrpcent, getrpcbyname, getrpcbynumber, setrpcent, or endrpcent **Subroutine**

Purpose

Accesses the /etc/rpc file.

Library

Standard C Library (libc.a)

Syntax

```
#include <netdb.h>
struct rpcent *getrpcent ()
struct rpcent *getrpcbyname ( Name)
char *Name:
struct rpcent *getrpcbynumber ( Number)
int Number;
void setrpcent (StayOpen)
int StayOpen
void endrpcent
```

Description

Attention: Do not use the getrpcent, getrpcbyname, getrpcbynumber, setrpcent, or endrpcent subroutine in a multithreaded environment.

Attention: The information returned by the **getrpcbyname**, and **getrpcbynumber** subroutines is stored in a static area and is overwritten on subsequent calls. Copy the information to save it.

The getprcbyname and getrpcbynumber subroutines each return a pointer to an object with the rpcent structure. This structure contains the broken-out fields of a line from the /etc/rpc file. The getprcbyname and getrpcbynumber subroutines searches the rpc file sequentially from the beginning of the file until it finds a matching RPC program name or number, or until it reaches the end of the file. The getrocent subroutine reads the next line of the file, opening the file if necessary.

The setrpcent subroutine opens and rewinds the /etc/rpc file. If the StayOpen parameter does not equal 0, the **rpc** file is not closed after a call to the **getrpcent** subroutine.

The **setrpcent** subroutine rewinds the **rpc** file. The **endrpcent** subroutine closes it.

The rpc file contains information about Remote Procedure Call (RPC) programs. The rpcent structure is in the /usr/include/netdb.h file and contains the following fields:

r name Contains the name of the server for an RPC program

r aliases Contains an alternate list of names for RPC programs. This list ends with a 0.

r number Contains a number associated with an RPC program.

Parameters

Name Specifies the name of a server for **rpc** program.

Number Specifies the **rpc** program number for service.

StayOpen Contains a value used to indicate whether to close the **rpc** file.

Return Values

These subroutines return a null pointer when they encounter the end of a file or an error.

Files

/etc/rpc Contains information about Remote Procedure Call (RPC) programs.

Related Information

Remote Procedure Call (RPC) for Programming in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs

getrusage, getrusage64, times, or vtimes Subroutine

Purpose

Displays information about resource use.

Libraries

getrusage, getrusage64, times: Standard C Library (libc.a)

vtimes: Berkeley Compatibility Library (libbsd.a)

Syntax

```
#include <sys/times.h>
#include <sys/resource.h>

int getrusage ( Who, RUsage)
int Who;
struct rusage *RUsage;

int getrusage64 ( Who, RUsage)
int Who;
struct rusage64 *RUsage;
#include <sys/types.h>
#include <sys/times.h>

clock_t times ( Buffer)
struct tms *Buffer;
#include <sys/times.h>
```

```
vtimes ( ParentVM, ChildVM)
struct vtimes *ParentVm, ChildVm;
```

Description

The getrusage subroutine displays information about how resources are used by the current process or all completed child processes.

When compiled in 64-bit mode, rusage counters are 64 bits. If getrusage is compiled in 32-bit mode, rusage counters are 32 bits. If the kernel's value of a usage counter has exceeded the capacity of the corresponding 32-bit rusage value being returned, the rusage value is set to INT MAX.

The getrusage64 subroutine can be called to make 64-bit rusage counters explicitly available in a 32-bit environment.

In AIX 5.1 and later, 64-bit quantities are also available to 64-bit applications through the getrusage() interface in the ru utime and ru stime fields of struct rusage.

The **times** subroutine fills the structure pointed to by the *Buffer* parameter with time-accounting information. All time values reported by the times subroutine are measured in terms of the number of clock ticks used. Applications should use sysconf (_SC_CLK_TCK) to determine the number of clock ticks per second.

The tms structure defined in the /usr/include/sys/times.h file contains the following fields:

```
time t tms utime;
time t tms stime;
time t tms cutime;
time t tms cstime;
```

This information is read from the calling process as well as from each completed child process for which the calling process executed a wait subroutine.

tms_utime	The CPU time used for executing instructions in the user space of the calling process
tms_stime	The CPU time used by the system on behalf of the calling process.
tms_cutime	The sum of the tms_utime and the tms_cutime values for all the child processes.
tms_cstime	The sum of the tms_stime and the tms_cstime values for all the child processes.

Note: The system measures time by counting clock interrupts. The precision of the values reported by the times subroutine depends on the rate at which the clock interrupts occur.

The **vtimes** subroutine is supported to provide compatibility with earlier programs.

The **vtimes** subroutine returns accounting information for the current process and for the completed child processes of the current process. Either the ParentVm parameter, the ChildVm parameter, or both may be 0. In that case, only the information for the nonzero pointers is returned.

After a call to the vtimes subroutine, each buffer contains information as defined by the contents of the /usr/include/sys/vtimes.h file.

Parameters

Who Specifies a value of RUSAGE THREAD, RUSAGE SELF, or RUSAGE CHILDREN.

RUsage

Points to a buffer described in the /usr/include/sys/resource.h file. The fields are interpreted as follows:

ru_utime

The total amount of time running in user mode.

ru stime

The total amount of time spent in the system executing on behalf of the processes.

ru maxrss

The maximum size, in kilobytes, of the used resident set size.

ru ixrss

An integral value indicating the amount of memory used by the text segment that was also shared among other processes. This value is expressed in units of kilobytes * seconds-of-execution and is calculated by adding the number of shared memory pages in use each time the internal system clock ticks, and then averaging over one-second intervals.

ru idrss

An integral value of the amount of unshared memory in the data segment of a process (expressed in units of kilobytes * seconds-of-execution).

ru minflt

The number of page faults serviced without any I/O activity. In this case, I/O activity is avoided by reclaiming a page frame from the list of pages awaiting reallocation.

ru majflt

The number of page faults serviced that required I/O activity.

ru nswap

The number of times a process was swapped out of main memory.

ru_inblock

The number of times the file system performed input.

ru oublock

The number of times the file system performed output.

Note: The numbers that the ru_inblock and ru_oublock fields display account for real I/O only; data supplied by the caching mechanism is charged only to the first process to read or write the data.

ru msgsnd

The number of IPC messages sent.

ru msgrcv

The number of IPC messages received.

ru nsignals

The number of signals delivered.

ru_nvcsw

The number of times a context switch resulted because a process voluntarily gave up the processor before its time slice was completed. This usually occurs while the process waits for availability of a resource.

ru nivcsw

The number of times a context switch resulted because a higher priority process ran or because the current process exceeded its time slice.

Buffer

ChildVm

Points to a **tms** structure.

Points to a vtimes structure that contains the accounting information for the current process. Points to a vtimes structure that contains the accounting information for the terminated child processes of the current process.

Return Values

Upon successful completion, the getrusage and getrusage64 subroutines return a value of 0. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Upon successful completion, the times subroutine returns the elapsed real time in units of ticks, whether profiling is enabled or disabled. This reference time does not change from one call of the times subroutine to another. If the times subroutine fails, it returns a value of -1 and sets the errno global variable to indicate the error.

Error Codes

The getrusage and getrusage64 subroutines do not run successfully if either of the following is true: **EINVAL** The Who parameter is not a valid value. **EFAULT** The address specified for RUsage is not valid. The times subroutine does not run successfully if the following is true:

EFAULT The address specified by the *buffer* parameter is not valid.

Related Information

The gettimer, settimer, restimer, stime, or time ("gettimer, settimer, restimer, stime, or time Subroutine" on page 375) subroutine, wait, waitpid, or wait3 subroutine.

Performance-Related Subroutines in AIX 5L Version 5.2 Performance Management Guide.

getroleattr, nextrole or putroleattr Subroutine

Purpose

Accesses the role information in the roles database.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
int getroleattr(Role, Attribute, Value, Type)
char *Role;
char *Attribute;
void *Value;
int Type;
char *nextrole(void)
int putroleattr(Role, Attribute, Value, Type)
char *Role;
char *Attribute;
void *Value;
int Type;
```

Description

The getroleattr subroutine reads a specified attribute from the role database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the putroleattr subroutine writes a specified attribute into the role database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the

putroleattr subroutine must be explicitly committed by calling the **putroleattr** subroutine with a Type parameter specifying SEC_COMMIT. Until all the data is committed, only the **getroleattr** subroutine within the process returns written data.

The **nextrole** subroutine returns the next role in a linear search of the role database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setroledb** and **endroledb** subroutines should be used to open and close the role database.

Parameters

Attribute

Specifies which attribute is read. The following possible attributes are defined in the usersec.h file:

S ROLELIST

List of roles included by this role. The attribute type is **SEC_LIST**.

S_AUTHORIZATIONS

List of authorizations included by this role. The attribute type is SEC_LIST.

S_GROUPS

List of groups required for this role. The attribute type is **SEC_LIST**.

S_SCREENS

List of SMIT screens required for this role. The attribute type is **SEC_LIST**.

S_VISIBILITY

Number value stating the visibility of the role. The attribute type is **SEC_INT**.

S MSGCAT

Message catalog file name. The attribute type is **SEC_CHAR**.

S MSGNUMBER

Message number within the catalog. The attribute type is **SEC_INT**.

Туре

Specifies the type of attribute expected. Valid types are defined in the usersec.h file and include:

SEC_INT

The format of the attribute is an integer.

For the **getroleattr** subroutine, the user should supply a pointer to a defined integer variable.

For the putroleattr subroutine, the user should supply an integer.

SEC CHAR

The format of the attribute is a null-terminated character string.

For the **getroleattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putroleattr** subroutine, the user should supply a character pointer.

SEC LIST

The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series must be an empty (zero character count) string.

For the **getroleattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putroleattr** subroutine, the user should supply a character pointer.

SEC_COMMIT

For the **putroleattr** subroutine, this value specified by itself indicates that changes to the named role are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no role is specified, the changes to all modified roles are committed to permanent storage.

SEC DELETE

The corresponding attribute is deleted from the database.

SEC_NEW

Updates the role database file with the new role name when using the **putroleattr** subroutine.

Value

Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Return Values

If successful, the **getroleattr** returns 0. Otherwise, a value of -1 is returned and the **errno** global variables is set to indicate the error.

Error Codes

Possible return codes are:

EACCES Access permission is denied for the data request.

ENOENT The specified *Role* parameter does not exist.

ENOATTRThe specified role attribute does not exist for this role.

The Attribute parameter does not contain one of the de

EINVAL The *Attribute* parameter does not contain one of the defined attributes or null.

The *Value* parameter does not point to a valid buffer or to valid data for this type of

attribute.

EPERM Operation is not permitted.

Related Information

The **getuserattr**, **nextusracl**, or **putusraclattr** ("getuserattr, IDtouser, nextuser, or putuserattr Subroutine" on page 382) subroutine, **setroledb**, or **endacldb** subroutine.

gets or fgets Subroutine

Purpose

Gets a string from a stream.

Library

Standard I/O Library (libc.a)

Syntax

```
#include <stdio.h>
char *gets ( String)
char *String;
char *fgets (String, Number, Stream)
char *String;
int Number:
FILE *Stream;
```

Description

The gets subroutine reads bytes from the standard input stream, stdin, into the array pointed to by the String parameter. It reads data until it reaches a new-line character or an end-of-file condition. If a new-line character stops the reading process, the gets subroutine discards the new-line character and terminates the string with a null character.

The **fgets** subroutine reads bytes from the data pointed to by the *Stream* parameter into the array pointed to by the String parameter. The fgets subroutine reads data up to the number of bytes specified by the Number parameter minus 1, or until it reads a new-line character and transfers that character to the String parameter, or until it encounters an end-of-file condition. The fgets subroutine then terminates the data string with a null character.

The first successful run of the **fgetc** ("getc, getchar, fgetc, or getw Subroutine" on page 300), **fgets**, fgetwc ("getwc, fgetwc, or getwchar Subroutine" on page 397), fgetws ("getws or fgetws Subroutine" on page 399), fread ("fread or fwrite Subroutine" on page 267), fscanf, getc ("getc, getchar, fgetc, or getw Subroutine" on page 300), getchar ("getc, getchar, fgetc, or getw Subroutine" on page 300), gets or scanf subroutine using a stream that returns data not supplied by a prior call to the ungetc or ungetwo subroutine marks the st atime field for update.

Parameters

String Points to a string to receive bytes.

Stream Points to the **FILE** structure of an open file.

Number Specifies the upper bound on the number of bytes to read.

Return Values

If the gets or fgets subroutine encounters the end of the file without reading any bytes, it transfers no bytes to the String parameter and returns a null pointer. If a read error occurs, the gets or fgets subroutine returns a null pointer and sets the errno global variable (errors are the same as for the fgetc ("getc, getchar, fgetc, or getw Subroutine" on page 300) subroutine). Otherwise, the gets or fgets subroutine returns the value of the String parameter.

Note: Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding the **SA RESTART** value.

Related Information

The **feof**, **ferror**, **clearerr**, or **fileno** ("feof, ferror, clearerr, or fileno Macro" on page 227) macro, **fopen**, freopen, or fdopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, fread ("fread or fwrite Subroutine" on page 267) subroutine, getc, getchar, fgetc, or getw ("getc, getchar, fgetc, or getw Subroutine" on page 300) subroutine, getwc, fgetwc, or getwchar ("getwc, fgetwc, or getwchar Subroutine" on page 397) subroutine, getws or fgetws ("getws or fgetws Subroutine" on page 399) subroutine, puts or fputs ("puts or fputs Subroutine" on page 994) subroutine, putws or fputws ("putws or fputws Subroutine" on page 997) subroutine, scanf, fscanf, or sscanf subroutine, ungetc or ungetwc subroutine.

List of String Manipulation Services, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getfsent_r, getfsspec_r, getfsfile_r, getfstype_r, setfsent_r, or endfsent r Subroutine

Purpose

Gets information about a file system.

Library

Thread-Safe C Library (libc_r.a)

Syntax

```
#include <fstab.h>
int getfsent_r (FSSent, FSFile, PassNo)
struct fstab * FSSent;
AFILE_t * FSFile;
int * PassNo;
int getfsspec_r (Special, FSSent, FSFile, PassNo)
const char * Special;
struct fstab *FSSent;
AFILE t *FSFile;
int *PassNo;
int getfsfile r (File, FSSent, FSFile, PassNo)
const char * File;
struct fstab *FSSent;
AFILE t *FSFile;
int *PassNo;
int getfstype r (Type, FSSent, FSFile, PassNo)
const char * Type;
struct fstab *FSSent;
AFILE t *FSFile;
int *PassNo;
int setfsent_r (FSFile, PassNo)
AFILE t * FSFile;
int *PassNo;
int endfsent r (FSFile)
AFILE_t *FSFile;
```

Description

The getfsent_r subroutine reads the next line of the /etc/filesystems file, opening it necessary.

The **setfsent_r** subroutine opens the **filesystems** file and positions to the first record.

The endfsent_r subroutine closes the filesystems file.

The **getfsspec_r** and **getfsfile_r** subroutines search sequentially from the beginning of the file until a matching special file name or file-system file name is found, or until the end of the file is encountered. The **getfstype_r** subroutine behaves similarly, matching on the file-system type field.

Programs using this subroutine must link to the libpthreads.a library.

Parameters

FSSent Points to a structure containing information about the file system. The FSSent parameter must be

allocated by the caller. It cannot be a null value.

FSFile Points to an attribute structure. The FSFile parameter is used to pass values between

subroutines.

PassNo Points to an integer. The **setfsent r** subroutine initializes the PassNo parameter.

Special Specifies a special file name to search for in the **filesystems** file.

File Specifies a file name to search for in the **filesystems** file.

Type Specifies a type to search for in the **filesystems** file.

Return Values

Indicates that the subroutine was successful.

-1 Indicates that the subroutine was not successful.

Files

/etc/filesystems Centralizes file-system characteristics.

Related Information

The **getvfsent**, **getvfsbytype**, **getvfsbyname**, **getvfsbyflag**, **setvfsent**, or **endvfsent** ("getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag, setvfsent, or endvfsent Subroutine" on page 395) subroutine.

The filesystems file in AIX 5L Version 5.2 Files Reference.

List of Multithread Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getsid Subroutine

Purpose

Returns the session ID of the calling process.

Library

(libc.a)

Syntax

#include <unistd.h>

pid_t getsid (pid_ t pid)

Description

The getsid subroutine returns the process group ID of the process that is the session leader of the process specified by pid. If pid is equal to pid_t subroutine, it specifies the calling process.

Parameters

pid A process ID of the process being queried.

Return Values

Upon successful completion, **getsid** subroutine returns the process group ID of the session leaded of the specified process. Otherwise, it returns (pid_t)-1 and set errno to indicate the error.

id The session ID of the requested process.

-1 Not successful and the errno global variable is set to one of the following error codes.

Error Codes

ESRCH There is no process with a process ID equal to pid.

The process specified by pid is not in the same session as the calling process. **EPERM**

ESRCH There is no process with a process ID equal to pid.

Related Information

The **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutines, fork ("fork, f_fork, or vfork Subroutine" on page 248) subroutines, getpid ("getpid, getpgrp, or getppid Subroutine" on page 345) subroutines, setpgid subroutines.

getssys Subroutine

Purpose

Reads a subsystem record.

Library

System Resource Controller Library (libsrc.a)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
int getssys( SubsystemName, SRCSubsystem)
char * SubsystemName;
struct SRCsubsys * SRCSubsystem;
```

Description

The **getssys** subroutine reads a subsystem record associated with the specified subsystem and returns the ODM record in the **SRCsubsys** structure.

The SRCsubsys structure is defined in the sys/srcobj.h file.

Parameters

SRCSubsystem Points to the **SRCsubsys** structure.

SubsystemName Specifies the name of the subsystem to be read.

Return Values

Upon successful completion, the **getssys** subroutine returns a value of 0. Otherwise, it returns a value of -1 and the **odmerrno** variable is set to indicate the error, or an SRC error code is returned.

Error Codes

If the getssys subroutine fails, the following is returned:

SRC_NOREC Subsystem name does not exist.

Files

/etc/objrepos/SRCsubsys

SRC Subsystem Configuration object class.

Related Information

The **addssys** ("addssys Subroutine" on page 19) subroutine, **delssys** ("delssys Subroutine" on page 171) subroutine, **getsubsvr** ("getsubsvr Subroutine" on page 371) subroutine.

Defining Your Subsystem to the SRC, List of SRC Subroutines, System Resource Controller (SRC) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getsubopt Subroutine

Purpose

Parse suboptions from a string.

Library

Standard C Library (libc.a)

Syntax

#include <stdlib.h>

int getsubopt (char **optionp,
char * const * tokens,
char ** valuep)

Description

The **getsubopt** subroutine parses suboptions in a flag parameter that were initially parsed by the **getopt** subroutine. These suboptions are separated by commas and may consist of either a single token, or a

token-value pair separated by an equal sign. Because commas delimit suboptions in the option string, they are not allowed to be part of the suboption or the value of a suboption, similarly, because the equal sign separates a token from its value, a token must not contain an equal sign.

The **getsubopt** subroutine takes the address of a pointer to the option string, a vector of possible tokens, and the address of a value string pointer. It returns the index of the token that matched the suboption in the input string or -1 if there was no match. If the option string at *optionp contains only one suboption, the **getsubopt** subroutine updates *optionp to point to the start of the next suboption. It the suboption has an associated value, the getsubopt subroutine updates *valuep to point to the value's first character. Otherwise it sets *valuep to a NULL pointer.

The token vector is organized as a series of pointers to strings. The end of the token vector is identified by a NULL pointer.

When the getsubopt subroutine returns, if *valuep is not a NULL pointer then the suboption processed included a value. The calling program may use this information to determine if the presence or lack of a value for this suboption is an error.

Additionally, when the **getsubopt** subroutine fails to match the suboption with the tokens in the *tokens* array, the calling program should decide if this is an error, or if the unrecognized option should be passed on to another program.

Return Values

The **getsubopt** subroutine returns the index of the matched token string, or -1 if no token strings were matched.

Related Information

The **getopt** ("getopt Subroutine" on page 337) subroutine.

getsubsvr Subroutine

Purpose

Reads a subsystem record.

Library

System Resource Controller Library (libsrc.a)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
int getsubsvr( SubserverName, SRCSubserver)
char *SubserverName;
struct SRCSubsvr *SRCSubserver;
```

Description

The **getsubsvr** subroutine reads a subsystem record associated with the specified subserver and returns the ODM record in the SRCsubsyr structure.

The SRCsubsvr structure is defined in the sys/srcobj.h file and includes the following fields:

```
char
                  sub_type[30];
```

char subsysname[30]; short sub_code;

Parameters

SRCSubserver Points to the SRCsubsvr structure. SubserverName Specifies the subserver to be read.

Return Values

Upon successful completion, the getsubsvr subroutine returns a value of 0. Otherwise, it returns a value of -1 and the **odmerrno** variable is set to indicate the error, or an SRC error code is returned.

Error Codes

If the **getsubsvr** subroutine fails, the following is returned:

SRC_NOREC The specified SRCsubsvr record does not exist.

Files

/etc/objrepos/SRCsubsvr

SRC Subserver Configuration object class.

Related Information

The **getssys** ("getssys Subroutine" on page 369) subroutine.

Defining Your Subsystem to the SRC, List of SRC Subroutines, System Resource Controller (SRC) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getthrds Subroutine

Purpose

Gets kernel thread table entries.

Library

Standard C library (libc.a)

Syntax

```
#include cinfo.h>
#include <sys/types.h>
getthrds ( ProcessIdentifier, ThreadBuffer, ThreadSize, IndexPointer, Count)
pid t ProcessIdentifier;
struct thrdsinfo *ThreadBuffer;
or struct thrdsinfo64 *ThreadBuffer;
int ThreadSize;
tid_t *IndexPointer;
int Count;
```

```
int
getthrds64 ( ProcessIdentifier, ThreadBuffer, ThreadSize, IndexPointer, Count)
pid t ProcessIdentifier;
struct thrdentry64 *ThreadBuffer;
int ThreadSize;
tid64 t *IndexPointer;
int Count;
```

Description

The getthrds subroutine returns information about kernel threads, including kernel thread table information defined by the thrdsinfo or thrdsinfo64 structure.

The **getthrds** subroutine retrieves up to *Count* kernel thread table entries, starting with the entry corresponding to the thread identifier indicated by IndexPointer, and places them in the array of thrdsinfo or thrdsinfo64, or thrdentry64 structures indicated by the *ThreadBuffer* parameter.

On return, the kernel thread identifier referenced by *IndexPointer* is updated to indicate the next kernel thread table entry to be retrieved. The getthrds subroutine returns the number of kernel thread table entries retrieved.

If the ProcessIdentifier parameter indicates a process identifier, only kernel threads belonging to that process are considered. If this parameter is set to -1, all kernel threads are considered.

The **getthrds** subroutine is normally called repeatedly in a loop, starting with a kernel thread identifier of zero, and looping until the return value is less than Count, indicating that there are no more entries to retrieve.

- 1. Do not use information from the procsinfo structure (see the getprocs ("getprocs Subroutine" on page 351) subroutine) to determine the value of the Count parameter; a process may create or destroy kernel threads in the interval between a call to getprocs and a subsequent call to getthrds.
- 2. The kernel thread table may change while the getthrds subroutine is accessing it. Returned entries will always be consistent, but since kernel threads can be created or destroyed while the getthrds subroutine is running, there is no quarantee that retrieved entries will still exist, or that all existing kernel threads have been retrieved.

When used in 32-bit mode, limits larger than can be represented in 32 bits are truncated to RLIM_INFINITY. Large values are truncated to INT_MAX. 64-bit applications are required to use getthrds64() and struct thrdentry64. Note that struct thrdentry64 contains the same information as struct thrdsinfo64 with the only difference being support for the 64-bit tid t and the 256-bit sigset t. Application developers are also encouraged to use **getthrds64()** in 32-bit applications to obtain 64-bit thread information as this interface provides the new, larger types. The getthrds() interface will still be supported for 32-bit applications using struct thrdsinfo or struct thrdsinfo64, but will not be available to 64-bit applications.

Parameters

ProcessIdentifier

Specifies the process identifier of the process whose kernel threads are to be retrieved. If this parameter is set to -1, all kernel threads in the kernel thread table are retrieved.

ThreadBuffer

Specifies the starting address of an array of thrdsinfo or thrdsinfo64, or thrdentry64 structures which will be filled in with kernel thread table entries. If a value of **NULL** is passed for this parameter, the getthrds subroutine scans the kernel thread table and sets return values as normal, but no kernel thread table entries are retrieved.

ThreadSize

Specifies the size of a single thrdsinfo, thrdsinfo64, or thrdentry64 structure.

IndexPointer

Specifies the address of a kernel thread identifier which indicates the required kernel thread table entry (this does not have to correspond to an existing kernel thread). A kernel thread identifier of zero selects the first entry in the table. The kernel thread identifier is updated to indicate the next entry to be retrieved.

Count Specifies the number of kernel thread table entries requested.

Return Value

If successful, the getthrds subroutine returns the number of kernel thread table entries retrieved; if this is less than the number requested, the end of the kernel thread table has been reached. Otherwise, a value of -1 is returned, and the errno global variable is set to indicate the error.

Error Codes

The **getthrds** subroutine fails if the following are true:

EINVAL The ThreadSize is invalid, or the IndexPointer parameter does not point to a valid kernel

thread identifier, or the Count parameter is not greater than zero.

ESRCH The process specified by the *ProcessIdentifier* parameter does not exist.

EFAULT The copy operation to one of the buffers failed.

Related Information

The getpid ("getpid, getpgrp, or getppid Subroutine" on page 345), getpgrp ("getpid, getpgrp, or getppid Subroutine" on page 345), or getppid ("getpid, getpgrp, or getppid Subroutine" on page 345) subroutines, the getprocs ("getprocs Subroutine" on page 351) subroutine.

The **ps** command.

gettimeofday, settimeofday, or ftime Subroutine

Purpose

Displays, gets and sets date and time.

Libraries

gettimeofday, settimeofday: Standard C Library (libc.a)

ftime: Berkeley Compatibility Library (libbsd.a)

Syntax

```
#include <sys/time.h>
int gettimeofday ( Tp, Tzp)
struct timeval *Tp;
void *Tzp;
int settimeofday (Tp, Tzp)
struct timeval *Tp;
struct timezone *Tzp;
#include <sys/types.h>
#include <sys/timeb.h>
int ftime (Tp)
struct timeb *Tp;
```

Description

Current Greenwich time and the current time zone are displayed with the gettimeofday subroutine, and set with the settimeofday subroutine. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware-dependent, and the time may be updated either continuously or in "ticks." If the Tzp parameter has a value of 0, the time zone information is not returned or set.

The Tp parameter returns a pointer to a timeval structure that contains the time since the epoch began in seconds and microseconds.

The timezone structure indicates both the local time zone (measured in minutes of time westward from Greenwich) and a flag that, if nonzero, indicates that daylight saving time applies locally during the appropriate part of the year.

In addition to the difference in timer granularity, the timezone structure distinguishes these subroutines from the POSIX gettimer and settimer subroutines, which deal strictly with Greenwich Mean Time.

The **ftime** subroutine fills in a structure pointed to by its argument, as defined by **<sys/timeb.h>**. The structure contains the time in seconds since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970, up to 1000 milliseconds of more-precise interval, the local timezone (measured in minutes of time westward from UTC), and a flag that, if nonzero, indicates that Daylight Saving time is in effect, and the values stored in the timeb structure have been adjusted accordingly.

Parameters

Pointer to a timeval structure, defined in the sys/time.h file.

Pointer to a timezone structure, defined in the sys/time.h file. Tzp

Return Values

If the subroutine succeeds, a value of 0 is returned. If an error occurs, a value of -1 is returned and errno is set to indicate the error.

Error Codes

If the settimeofday subroutine is unsuccessful, the errno value is set to EPERM to indicate that the process's effective user ID does not have root user authority.

No errors are defined for the **gettimeofday** or **ftime** subroutine.

gettimer, settimer, restimer, stime, or time Subroutine

Purpose

Gets or sets the current value for the specified systemwide timer.

Library

Standard C Library (libc.a)

Syntax

#include <sys/time.h> #include <sys/types.h>

```
int gettimer( TimerType, Value)
timer t TimerType;
struct timestruc t * Value;
#include <sys/timers.h>
#include <sys/types.h>
int gettimer( TimerType, Value)
timer t TimerType;
struct itimerspec * Value;
int settimer(TimerType, TimePointer)
int TimerType;
const struct timestruc t *TimePointer;
int restimer(TimerType, Resolution, MaximumValue)
int TimerType;
struct timestruc t *Resolution, *MaximumValue;
int stime( Tp)
long *Tp;
#include <sys/types.h>
time t time(Tp)
time_t *Tp;
```

Description

The **settimer** subroutine is used to set the current value of the *TimePointer* parameter for the systemwide timer, specified by the *TimerType* parameter.

When the gettimer subroutine is used with the function prototype in sys/timers.h, then except for the parameters, the gettimer subroutine is identical to the getinterval ("getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329) subroutine. Use of the getinterval subroutine is recommended, unless the gettimer subroutine is required for a standards-conformant application. The description and semantics of the gettimer subroutine are subject to change between releases, pending changes in the draft standard upon which the current gettimer subroutine description is based.

When the gettimer subroutine is used with the function prototype in /sys/timers.h, the gettimer subroutine returns an **itimerspec** structure to the pointer specified by the *Value* parameter. The **it value** member of the itimerspec structure represents the amount of time in the current interval before the timer (specified by the *TimerType* parameter) expires, or a zero interval if the timer is disabled. The members of the pointer specified by the Value parameter are subject to the resolution of the timer.

When the gettimer subroutine is used with the function prototype in sys/time.h, the gettimer subroutine returns a timestruc structure to the pointer specified by the Value parameter. This structure holds the current value of the system wide timer specified by the Value parameter.

The resolution of any timer can be obtained by the **restimer** subroutine. The *Resolution* parameter represents the resolution of the specified timer. The MaximumValue parameter represents the maximum possible timer value. The value of these parameters are the resolution accepted by the settimer subroutine.

Note: If a nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of less than 10 milliseconds), the timer request is raised to 10 milliseconds.

The **time** subroutine returns the time in seconds since the Epoch (that is, 00:00:00 GMT, January 1, 1970). The Tp parameter points to an area where the return value is also stored. If the Tp parameter is a null pointer, no value is stored.

The stime subroutine is implemented to provide compatibility with older AIX, AT&T System V, and BSD systems. It calls the **settimer** subroutine using the **TIMEOFDAY** timer.

Parameters

Value Points to a structure of type **itimerspec**.

TimerType Specifies the systemwide timer:

TIMEOFDAY

(POSIX system clock timer) This timer represents the time-of-day clock for the system. For this timer, the values returned by the gettimer subroutine and specified by the settimer subroutine represent the amount of time since

00:00:00 GMT, January 1, 1970.

TimePointer | Points to a structure of type **struct timestruc_t**.

Resolution The resolution of a specified timer. MaximumValue The maximum possible timer value.

Points to a structure containing the time in seconds. Тp

Return Values

The **gettimer**, **settimer**, **restimer**, and **stime** subroutines return a value of 0 (zero) if the call is successful. A return value of -1 indicates an error occurred, and errno is set.

The time subroutine returns the value of time in seconds since Epoch. Otherwise, a value of ((time t) - 1) is returned and the **errno** global variable is set to indicate the error.

Error Codes

If an error occurs in the gettimer, settimer, restimer, or stime subroutine, a return value of - 1 is received and the **errno** global variable is set to one of the following error codes:

EINVAL The TimerType parameter does not specify a known systemwide timer, or the TimePointer

parameter of the settimer subroutine is outside the range for the specified systemwide timer.

EFAULT A parameter address referenced memory that was not valid.

An error occurred while accessing the timer device. EIO

EPERM The requesting process does not have the appropriate privilege to set the specified timer.

If the time subroutine is unsuccessful, a return value of -1 is received and the errno global variable is set to the following:

EFAULT A parameter address referenced memory that was not valid.

Related Information

The asctime ("ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine" on page 163) subroutine, clock ("clock Subroutine" on page 140) subroutine, ctime ("ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine" on page 163) subroutine, difftime ("ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine" on page 163) subroutine, getinterval ("getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329) subroutine, gmtime ("ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine" on page 163) subroutine, localtime ("ctime, localtime, mktime, mktime, difftime, asctime, or tzset Subroutine" on page 163

page 163) subroutine, **mktime** ("ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine" on page 163) subroutine, strftime subroutine, strptime subroutine, utime subroutine.

List of Time Data Manipulation Services in AIX 5L Version 5.2 System Management Concepts: Operating System and Devices.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

gettimerid Subroutine

Purpose

Allocates a per-process interval timer.

Library

Standard C Library (libc.a)

Syntax

#include <sys/time.h> #include <sys/events.h>

timer_t gettimerid(TimerType, NotifyType) int TimerType; int NotifyType;

Description

The gettimerid subroutine is used to allocate a per-process interval timer based on the timer with the given timer type. The unique ID is used to identify the interval timer in interval timer requests. (See getinterval subroutine). The particular timer type, the TimerType parameter, is defined in the sys/time.h file and can identify either a systemwide timer or a per-process timer. The mechanism by which the process is to be notified of the expiration of the timer event is the NotifyType parameter, which is defined in the sys/events.h file.

The *TimerType* parameter represents one of the following timer types:

TIMEOFDAY (POSIX system clock timer) This timer represents the time-of-day clock for the

system. For this timer, the values returned by the gettimer subroutine and specified by the settimer subroutine represent the amount of time since 00:00:00

GMT, January 1, 1970, in nanoseconds.

TIMERID_ALRM (Alarm timer) This timer schedules the delivery of a SIGALRM signal at a timer

specified in the call to the settimer subroutine.

TIMERID REAL (Real-time timer) The real-time timer decrements in real time. A SIGALRM signal

is delivered when this timer expires.

TIMERID_VIRTUAL (Virtual timer) The virtual timer decrements in process virtual time, it runs only

when the process is executing in user mode. A SIGVTALRM signal is delivered

when it expires.

TIMERID PROF (Profiling timer) The profiling timer decrements both when running in user mode

> and when the system is running for the process. It is designed to be used by processes to profile their execution statistically. A SIGPROF signal is delivered

when the profiling timer expires.

Interval timers with a notification value of **DELIVERY_SIGNAL** are inherited across an **exec** subroutine.

Parameters

NotifyType Notifies the process of the expiration of the timer event. *TimerType* Identifies either a systemwide timer or a per-process timer.

Return Values

If the **gettimerid** subroutine succeeds, it returns a **timer t** structure that can be passed to the per-process interval timer subroutines, such as the getinterval subroutine. If an error occurs, the value -1 is returned and errno is set.

Error Codes

If the **gettimerid** subroutine fails, the value -1 is returned and **errno** is set to one of the following error codes:

EAGAIN The calling process has already allocated all of the interval timers associated with the specified

timer type for this implementation.

EINVAL The specified timer type is not defined.

Related Information

The **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutine, fork ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, getinterval, incinterval, absinterval, resinc, or resabs ("getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329) subroutine, gettimer, settimer, or restimer ("gettimer, settimer, restimer, stime, or time Subroutine" on page 375) subroutine, reltimerid subroutine.

List of Time Data Manipulation Services in AIX 5L Version 5.2 System Management Concepts: Operating System and Devices.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getttyent, getttynam, setttyent, or endttyent Subroutine

Purpose

Gets a tty description file entry.

Library

Standard C Library (libc.a)

Syntax

```
#include <ttyent.h>
```

```
struct ttyent *getttyent()
struct ttyent *getttynam( Name)
char *Name;
void setttyent()
void endttyent()
```

Description

Attention: Do not use the **getttyent**, **getttynam**, **setttyent**, or **endttyent** subroutine in a multithreaded environment.

The **getttyent** and **getttynam** subroutines each return a pointer to an object with the **ttyent** structure. This structure contains the broken-out fields of a line from the tty description file. The **ttyent** structure is in the **/usr/include/sys/ttyent.h** file and contains the following fields:

tty_name The name of the character special file in the /dev directory. The character special file must

reside in the /dev directory.

ty getty

The command that is called by the init process to initialize tty line characteristics. This is

usually the getty command, but any arbitrary command can be used. A typical use is to initiate

a terminal emulator in a window system.

ty_type The name of the default terminal type connected to this tty line. This is typically a name from

the termcap database. The TERM environment variable is initialized with this name by the

getty or login command.

ty status A mask of bit fields that indicate various actions to be allowed on this tty line. The following is a

description of each flag:

TTY_ON

Enables logins (that is, the init process starts the specified getty command on this

entry).

TTY_SECURE

Allows a user with root user authority to log in to this terminal. The TTY_ON flag must

be included.

ty_window The command to execute for a window system associated with the line. The window system is

started before the command specified in the ty getty field is executed. If none is specified,

this is null.

ty comment The trailing comment field. A leading delimiter and white space is removed.

The **getttyent** subroutine reads the next line from the tty file, opening the file if necessary. The **setttyent** subroutine rewinds the file. The **endttyent** subroutine closes it.

The **getttynam** subroutine searches from the beginning of the file until a matching name (specified by the *Name* parameter) is found (or until the EOF is encountered).

Parameters

Name Specifies the name of a tty description file.

Return Values

These subroutines return a null pointer when they encounter an EOF (end-of-file) character or an error.

Files

/usr/lib/libodm.a Specifies the ODM (Object Data Manager) library.

/usr/lib/libcfg.a Archives device configuration subroutines.

/etc/termcap Defines terminal capabilities.

Related Information

The ttyslot subroutine.

The getty command, init command, login command.

List of Files and Directories Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getuid, geteuid, or getuidx Subroutine

Purpose

Gets the real or effective user ID of the current process.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/types.h>
#include <unistd.h>
uid t getuid(void)
uid_t geteuid(void)
#include <id.h>
uid_t getuidx (int type);
```

Description

The getuid subroutine returns the real user ID of the current process. The geteuid subroutine returns the effective user ID of the current process.

The **getuidx** subroutine returns the user ID indicated by the *type* parameter of the calling process.

These subroutines are part of Base Operating System (BOS) Runtime.

Return Values

The getuid, geteuid and getuidx subroutines return the corresponding user ID. The getuid and geteuid subroutines always succeed.

The getuidx subroutine will return -1 and set the global errno variable to EINVAL if the type parameter is not one of ID REAL, ID EFFECTIVE, ID SAVED or ID LOGIN.

Parameters

type

Specifies the user ID to get. Must be one of ID_REAL (real user ID), ID_EFFECTIVE (effective user ID), ID_SAVED (saved set-user ID) or ID_LOGIN (login user ID).

Error Codes

If the **getuidx** subroutine fails the following is returned:

EINVAL Indicates the value of the type parameter is invalid.

Related Information

The **setuid** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getuinfo Subroutine

Purpose

Finds a value associated with a user.

Library

Standard C Library (libc.a)

Syntax

```
char *getuinfo ( Name)
char *Name:
```

Description

The getuinfo subroutine finds a value associated with a user. This subroutine searches a user information buffer for a string of the form Name=Value and returns a pointer to the Value substring if the Name value is found. A null value is returned if the Name value is not found.

The **INuibp** global variable points to the user information buffer:

```
extern char *INuibp;
```

This variable is initialized to a null value.

If the **INuibp** global variable is null when the **getuinfo** subroutine is called, the **usrinfo** subroutine is called to read user information from the kernel into a local buffer. The INUuibp is set to the address of the local buffer. If the INuibp external variable is not set, the usrinfo subroutine is automatically called the first time the **getuinfo** subroutine is called.

Parameter

Name

Specifies a user name.

Related Information

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getuserattr, IDtouser, nextuser, or putuserattr Subroutine

Purpose

Accesses the user information in the user database.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
```

```
int getuserattr (User, Attribute, Value, Type)
char * User;
```

```
char * Attribute;
void * Value;
int Type;
char *IDtouser( UID)
uid t UID;
char *nextuser ( Mode, Argument)
int Mode, Argument;
int putuserattr (User, Attribute, Value, Type)
char *User:
char *Attribute;
void *Value;
int Type;
```

Description

Attention: These subroutines and the setpwent and setgrent subroutines should not be used simultaneously. The results can be unpredictable.

These subroutines access user information. Because of their greater granularity and extensibility, you should use them instead of the **getpwent** routines.

The **getuserattr** subroutine reads a specified attribute from the user database. If the database is not already open, this subroutine does an implicit open for reading. A call to the getuserattr subroutine for every new user verifies that the user exists.

Similarly, the putuserattr subroutine writes a specified attribute into the user database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the putuserattr subroutine must be explicitly committed by calling the putuserattr subroutine with a Type parameter specifying SEC COMMIT. Until all the data is committed, only these subroutines within the process return written data.

New entries in the user and group databases must first be created by invoking putuserattr with the **SEC NEW** type.

The **IDtouser** subroutine translates a user ID into a user name.

The **nextuser** subroutine returns the next user in a linear search of the user database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setuserdb** and **enduserdb** subroutines should be used to open and close the user database.

The **enduserdb** subroutine frees all memory allocated by the **getuserattr** subroutine.

Parameters

Argument

Presently unused and must be specified as null.

Attribute

Specifies which attribute is read. The following possible attributes are defined in the usersec.h file:

S_ID User ID. The attribute type is **SEC_INT**.

S PGRP

Principle group name. The attribute type is **SEC_CHAR**.

S GROUPS

Groups to which the user belongs. The attribute type is **SEC_LIST**.

S ADMGROUPS

Groups for which the user is an administrator. The attribute type is **SEC_LIST**.

S ADMIN

Administrative status of a user. The attribute type is **SEC_BOOL**.

S AUDITCLASSES

Audit classes to which the user belongs. The attribute type is **SEC_LIST**.

S AUTHSYSTEM

Defines the user's authentication method. The attribute type is **SEC_CHAR**.

S HOME

Home directory. The attribute type is **SEC_CHAR**.

S SHELL

Initial program run by a user. The attribute type is **SEC_CHAR**.

S GECOS

Personal information for a user. The attribute type is **SEC_CHAR**.

S USRENV

User-state environment variables. The attribute type is SEC_LIST.

S SYSENV

Protected-state environment variables. The attribute type is **SEC_LIST**.

S LOGINCHK

Specifies whether the user account can be used for local logins. The attribute type is **SEC_BOOL**.

S_HISTEXPIRE

Defines the period of time (in weeks) that a user cannot reuse a password. The attribute type is **SEC_INT**.

S HISTSIZE

Specifies the number of previous passwords that the user cannot reuse. The attribute type is **SEC INT**.

S_MAXREPEAT

Defines the maximum number of times a user can repeat a character in a new password. The attribute type is **SEC_INT**.

S MINAGE

Defines the minimum age in weeks that the user's password must exist before the user can change it. The attribute type is **SEC_INT**.

S_PWDCHECKS

Defines the password restriction methods for this account. The attribute type is SEC_LIST.

S MINALPHA

Defines the minimum number of alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S MINDIFF

Defines the minimum number of characters required in a new password that were not in the old password. The attribute type is **SEC_INT**.

S MINLEN

Defines the minimum length of a user's password. The attribute type is **SEC_INT**.

S MINOTHER

Defines the minimum number of non-alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S_DICTIONLIST

Defines the password dictionaries for this account. The attribute type is **SEC_LIST**.

S SUCHK

Specifies whether the user account can be accessed with the su command. Type SEC_BOOL.

S REGISTRY

Defines the user's authentication registry. The attribute type is **SEC_CHAR**.

S RLOGINCHK

Specifies whether the user account can be used for remote logins using the telnet or **rlogin** commands. The attribute type is **SEC_BOOL**.

S DAEMONCHK

Specifies whether the user account can be used for daemon execution of programs and subsystems using the **cron** daemon or **src**. The attribute type is **SEC_BOOL**.

S_TPATH

Defines how the account may be used on the trusted path. The attribute type is **SEC CHAR**. This attribute must be one of the following values:

nosak The secure attention key is not enabled for this account.

notsh The trusted shell cannot be accessed from this account.

always

This account may only run trusted programs.

Normal trusted-path processing applies. on

S TTYS

List of ttys that can or cannot be used to access this account. The attribute type is SEC_LIST.

S_SUGROUPS

Groups that can or cannot access this account. The attribute type is SEC_LIST.

S_EXPIRATION

Expiration date for this account is a string in the form MMDDhhmmyy, where MM is the month, DD is the day, hh is the hour in 0 to 24 hour notation, mm is the minutes past the hour, and yy is the last two digits of the year. The attribute type is **SEC_CHAR**.

S AUTH1

Primary authentication methods for this account. The attribute type is SEC_LIST.

S AUTH2

Secondary authentication methods for this account. The attribute type is SEC_LIST.

S UFSIZE

Process file size soft limit. The attribute type is **SEC_INT**.

S UCPU

Process CPU time soft limit. The attribute type is **SEC INT**.

S UDATA

Process data segment size soft limit. The attribute type is **SEC_INT**.

S USTACK

Process stack segment size soft limit. Type: **SEC INT**.

S URSS

Process real memory size soft limit. Type: **SEC INT**.

S UCORE

Process core file size soft limit. The attribute type is **SEC_INT**.

S_UNOFILE

Process file descriptor table size soft limit. The attribute type is **SEC_INT**.

S PWD

Specifies the value of the passwd field in the **/etc/passwd** file. The attribute type is **SEC CHAR**.

S UMASK

File creation mask for a user. The attribute type is **SEC_INT**.

S LOCKED

Specifies whether the user's account can be logged into. The attribute type is **SEC BOOL**.

S ROLES

Defines the administrative roles for this account. The attribute type is **SEC_LIST**.

S UFSIZE HARD

Process file size hard limit. The attribute type is **SEC_INT**.

S UCPU HARD

Process CPU time hard limit. The attribute type is **SEC_INT**.

S UDATA HARD

Process data segment size hard limit. The attribute type is **SEC_INT**.

S USREXPORT

Specifies if the DCE registry can overwrite the local user information with the DCE user information during a DCE export operation. The attribute type is **SEC_BOOL**.

S_USTACK_HARD

Process stack segment size hard limit. Type: **SEC_INT**.

S URSS HARD

Process real memory size hard limit. Type: **SEC_INT**.

S_UCORE_HARD

Process core file size hard limit. The attribute type is **SEC_INT**.

S UNOFILE_HARD

Process file descriptor table size hard limit. The attribute type is **SEC_INT**.

Note: These values are string constants that should be used by applications both for convenience and to permit optimization in latter implementations. Additional user-defined attributes may be used and will be stored in the format specified by the *Type* parameter.

Mode Specifies the search mode. This parameter can be used to delimit the search to one or more user credentials databases. Specifying a non-null Mode value also implicitly rewinds the search. A null Mode value continues the search sequentially through the database. This parameter must include one of the following values specified as a bit mask; these are defined in the **usersec.h** file:

S LOCAL

Locally defined users are included in the search.

S_SYSTEM

All credentials servers for the system are searched.

Type Specifies the type of attribute expected. Valid types are defined in the usersec.h file and include:

SEC_INT

The format of the attribute is an integer.

For the getuserattr subroutine, the user should supply a pointer to a defined integer variable. For the **putuserattr** subroutine, the user should supply an integer.

SEC_CHAR

The format of the attribute is a null-terminated character string.

For the **getuserattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the putuserattr subroutine, the user should supply a character pointer.

SEC_LIST

The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters.

For the getuserattr subroutine, the user should supply a pointer to a defined character pointer variable. For the putuserattr subroutine, the user should supply a character pointer.

SEC BOOL

The format of the attribute from **getuserattr** is an integer with the value of either 0 (false) or 1 (true). The format of the attribute for putuserattr is a null-terminated string containing one of the following strings: true, false, yes, no, always, or never.

For the **getuserattr** subroutine, the user should supply a pointer to a defined integer variable. For the putuserattr subroutine, the user should supply a character pointer.

SEC_COMMIT

For the **putuserattr** subroutine, this value specified by itself indicates that changes to the named user are to be committed to permanent storage. The Attribute and Value parameters are ignored. If no user is specified, the changes to all modified users are committed to permanent storage.

SEC DELETE

The corresponding attribute is deleted from the database.

SEC NEW

Updates all the user database files with the new user name when using the putuserattr subroutine.

UID Specifies the user ID to be translated into a user name.

User Specifies the name of the user for which an attribute is to be read.

Value Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the Attribute and Type parameters. See the Type parameter for more details.

Security

Files Accessed:

Mode	File
rw	/etc/passwd
rw	/etc/group
rw	/etc/security/user

Mode File

 rw
 /etc/security/limits

 rw
 /etc/security/group

 rw
 /etc/security/environ

Return Values

If successful, the **getuserattr** subroutine with the **S_LOGINCHK** or **S_RLOGINCHK** attribute specified and the **putuserattr** subroutine return 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error. For all other attributes, the **getuserattr** subroutine returns 0.

If successful, the **IDtouser** and **nextuser** subroutines return a character pointer to a buffer containing the requested user name. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

If any of these subroutines fail, the following is returned:

EACCES Access permission is denied for the data request.

If the getuserattr and putuserattr subroutines fail, one or more of the following is returned:

ENOENT The specified *User* parameter does not exist.

EINVAL The Attribute parameter does not contain one of the defined attributes or null.

EINVAL The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.

Limited testing is possible and all errors may not be detected.

EPERM Operation is not permitted.

ENOATTR The specified attribute is not defined for this user.

If the **IDtouser** subroutine fails, one or more of the following is returned:

ENOENT The specified *User* parameter does not exist

If the **nextuser** subroutine fails, one or more of the following is returned:

EINVAL The *Mode* parameter is not one of null, S_LOCAL, or S_SYSTEM.

EINVAL The *Argument* parameter is not null. **ENOENT** The end of the search was reached.

Files

/etc/passwd Contains user IDs.

Related Information

The "getgroupattr, IDtogroup, nextgroup, or putgroupattr Subroutine" on page 322, "getuserpw, putuserpw, or putuserpwhist Subroutine" on page 389, **setpwdb** subroutine, **setuserdb** subroutine.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

GetUserAuths Subroutine

Purpose

Accesses the set of authorizations of a user.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
char *GetUserAuths(void);
```

Description

The GetUserAuths subroutine returns the list of authorizations associated with the real user ID and group set of the process. By default, the ALL authorization is returned for the root user.

Return Values

If successful, the GetUserAuths subroutine returns a list of authorizations associated with the user. The format of the list is a series of concatenated strings, each null-terminated. A null string terminates the list. Otherwise, a null pointer is returned and the errno global variable is set to indicate the error.

getuserpw, putuserpw, or putuserpwhist Subroutine

Purpose

Accesses the user authentication data.

Library

Security Library (libc.a)

Syntax

```
#include <userpw.h>
struct userpw *getuserpw ( User)
char *User;
int putuserpw ( Password)
struct userpw *Password;
int putuserpwhist ( Password, Message)
struct userpw *Password;
char **Message;
```

Description

These subroutines may be used to access user authentication information. Because of their greater granularity and extensibility, you should use them instead of the getpwent routines.

The getuserpw subroutine reads the user's locally defined password information. If the setpwdb subroutine has not been called, the getuserpw subroutine will call it as setpwdb (\$ READ). This can cause problems if the **putuserpw** subroutine is called later in the program.

The **putuserpw** subroutine updates or creates a locally defined password information stanza in the **/etc/security/passwd** file. The password entry created by the **putuserpw** subroutine is used only if there is an! (exclamation point) in the **/etc/passwd** file's password field. The user application can use the **putuserattr** subroutine to add an! to this field.

The **putuserpw** subroutine will open the authentication database read/write if no other access has taken place, but the program should call setpwdb (S READ | S WRITE) before calling the **putuserpw** subroutine.

The **putuserpwhist** subroutine updates or creates a locally defined password information stanza in the **etc/security/passwd** file. The subroutine also manages a database of previous passwords used for password reuse restriction checking. It is recommended to use the **putuserpwhist** subroutine, rather than the **putuserpw** subroutine, to ensure the password is added to the password history database.

Parameters

Password

Specifies the password structure used to update the password information for this user. This structure is defined in the **userpw.h** file and contains the following members:

upw name

Specifies the user's name. (The first eight characters must be unique, since longer names are truncated.)

upw passwd

Specifies the user's password.

upw_lastupdate

Specifies the time, in seconds, since the epoch (that is, 00:00:00 GMT, January 1, 1970), when the password was last updated.

upw_flags

Specifies attributes of the password. This member is a bit mask of one or more of the following values, defined in the **userpw.h** file.

PW NOCHECK

Specifies that new passwords need not meet password restrictions in effect for the system.

PW_ADMCHG

Specifies that the password was last set by an administrator and must be changed at the next successful use of the **login** or **su** command.

PW ADMIN

Specifies that password information for this user may only be changed by the root user.

Message Indicates a message that specifies an error occurred while updating the password history database.

Upon return, the value is either a pointer to a valid string within the memory allocated storage or a

null pointer.

Specifies the name of the user for which password information is read. (The first eight characters

must be unique, since longer names are truncated.)

Security

User

Files Accessed:

Mode File

rw /etc/security/passwd

Return Values

If successful, the getuserpw subroutine returns a valid pointer to a pw structure. Otherwise, a null pointer is returned and the errno global variable is set to indicate the error.

If successful, the putuserpwhist subroutine returns a value of 0. If the subroutine failed to update or create a locally defined password information stanza in the /etc/security/ passwd file, the putuserpwhist subroutine returns a nonzero value. If the subroutine was unable to update the password history database, a message is returned in the Message parameter and a return code of 0 is returned.

Error Codes

If the **getuserpw**, putuserpw, and putuserpwhist subroutines fail if one of the following values is true:

ENOATTR The user has an entry in the /etc/security/password file but does not have a Password

ENOENT The user does not have an entry in the /etc/security/passwd file.

Subroutines invoked by the **getuserpw**, **putuserpw**, or **putuserpwhist** subroutines can also set errors.

Files

/etc/security/passwd

Contains user passwords.

Related Information

The "getgroupattr, IDtogroup, nextgroup, or putgroupattr Subroutine" on page 322, "getuserattr, IDtouser, nextuser, or putuserattr Subroutine" on page 382, setpwdb or endpwdb subroutine, setuserdb subroutine.

List of Security and Auditing Subroutines and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getusraclattr, nextusracl or putusraclattr Subroutine

Purpose

Accesses the user screen information in the SMIT ACL database.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
int getusraclattr(User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
char *nextusracl(void)
int putusraclattr(User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```

Description

The **getusraclattr** subroutine reads a specified user attribute from the SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the **putusraclattr** subroutine writes a specified attribute into the user SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putusraclattr** subroutine must be explicitly committed by calling the **putusraclattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only the **getusraclattr** subroutine within the process returns written data.

The **nextusracl** subroutine returns the next user in a linear search of the user SMIT ACL database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setacldb** and **endacldb** subroutines should be used to open and close the database.

Parameters

Attribute

Specifies which attribute is read. The following possible attributes are defined in the usersec.h file:

S_SCREENS

String of SMIT screens. The attribute type is **SEC_LIST**.

S_ACLMODE

String specifying the SMIT ACL database search scope. The attribute type is **SEC_CHAR**.

S FUNCMODE

String specifying the databases to be searched. The attribute type is **SEC_CHAR**. Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include:

Туре

SEC CHAR

The format of the attribute is a null-terminated character string.

For the **getusraclattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putusraclattr** subroutine, the user should supply a character pointer.

SEC_LIST

The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series must be an empty (zero character count) string.

For the **getusraclattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putusraclattr** subroutine, the user should supply a character pointer.

SEC COMMIT

For the **putusraclattr** subroutine, this value specified by itself indicates that changes to the named user are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no user is specified, the changes to all modified users are committed to permanent storage.

SEC_DELETE

The corresponding attribute is deleted from the user SMIT ACL database.

SEC_NEW

Updates the user SMIT ACL database file with the new user name when using the **putusraclattr** subroutine.

Value

Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Return Values

If successful, the getusraclattr returns 0. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

Possible return codes are:

EACCES Access permission is denied for the data request.

ENOENT The specified User parameter does not exist or the attribute is not defined for this user.

ENOATTR The specified user attribute does not exist for this user.

EINVAL The Attribute parameter does not contain one of the defined attributes or null.

EINVAL The Value parameter does not point to a valid buffer or to valid data for this type of attribute.

EPERM Operation is not permitted.

Related Information

The getgrpaclattr, nextgrpacl, or putgrpaclattr ("getgrpaclattr, nextgrpacl, or putgrpaclattr Subroutine" on page 326) subroutine, setacldb, or endacldb subroutine.

getutent, getutid, getutline, pututline, setutent, endutent, or utmpname **Subroutine**

Purpose

Accesses utmp file entries.

Library

Standard C Library (libc.a)

Syntax

```
#include <utmp.h>
struct utmp *getutent ( )
struct utmp *getutid ( ID)
struct utmp *ID;
struct utmp *getutline ( Line)
struct utmp *Line;
void pututline ( Utmp)
struct utmp *Utmp;
void setutent ( )
void endutent ( )
void utmpname ( File)
char *File;
```

Description

The **getutent**, **getutid**, and **getutline** subroutines return a pointer to a structure of the following type:

```
struct utmp
> {
    >
>
>
      >
    short ut_type;
>
>
>
       struct exit status
          short e termination; /* Process termination status */
>
>
                        /* Process exit status */
          short e exit;
       }
       ut exit;
                         /* The exit status of a process
                         /* marked as DEAD_PROCESS. */
       char ut host[256];
                         /* host name */
>
       int __dbl_word_pad;
                         /* for double word alignment */
       int __reservedA[2];
       int __reservedV[6];
> };
```

The getutent subroutine reads the next entry from a utmp-like file. If the file is not open, this subroutine opens it. If the end of the file is reached, the **getutent** subroutine fails.

The pututline subroutine writes the supplied *Utmp* parameter structure into the utmp file. It is assumed that the user of the pututline subroutine has searched for the proper entry point using one of the getut subroutines. If not, the pututline subroutine calls getutid to search forward for the proper place. If so, pututline does not search. If the pututline subroutine does not find a matching slot for the entry, it adds a new entry to the end of the file.

The **setutent** subroutine resets the input stream to the beginning of the file. Issue a **setuid** call before each search for a new entry if you want to examine the entire file.

The **endutent** subroutine closes the file currently open.

The **utmpname** subroutine changes the name of a file to be examined from **/etc/utmp** to any other file. The name specified is usually /var/adm/wtmp. If the specified file does not exist, no indication is given. You are not aware of this fact until your first attempt to reference the file. The utmpname subroutine does not open the file. It closes the old file, if currently open, and saves the new file name.

The most current entry is saved in a static structure. To make multiple accesses, you must copy or use the structure between each access. The getutid and getutline subroutines examine the static structure first. If the contents of the static structure match what they are searching for, they do not read the utmp file. Therefore, you must fill the static structure with zeros after each use if you want to use these subroutines to search for multiple occurrences.

If the **pututline** subroutine finds that it is not already at the correct place in the file, the implicit read it performs does not overwrite the contents of the static structure returned by the getutent subroutine, the getuid subroutine, or the getutline subroutine. This allows you to get an entry with one of these subroutines, modify the structure, and pass the pointer back to the pututline subroutine for writing.

These subroutines use buffered standard I/O for input. However, the **pututline** subroutine uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the utmp and wtmp files.

Parameters

ID If you specify a type of RUN_LVL, BOOT_TIME, OLD_TIME, or NEW_TIME in the ID parameter,

the getutid subroutine searches forward from the current point in the utmp file until an entry with

a ut_type matching ID->ut_type is found.

If you specify a type of INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or

DEAD_PROCESS in the *ID* parameter, the **getutid** subroutine returns a pointer to the first entry whose type is one of these four and whose ut id field matches Id->ut id. If the end of the file is

reached without a match, the getutid subroutine fails.

Line The **getutline** subroutine searches forward from the current point in the **utmp** file until it finds an

entry of type LOGIN_PROCESS or USER_PROCESS that also has a ut_line string matching the

Line->ut_line parameter string. If the end of file is reached without a match, the **getutline**

subroutine fails.

Utmp Points to the **utmp** structure.

File Specifies the name of the file to be examined.

Return Values

These subroutines fail and return a null pointer if a read or write fails due to a permission conflict or because the end of the file is reached.

Files

/etc/utmp /var/adm/wtmp Path to the **utmp** file, which contains a record of users logged into the system. Path to the **wtmp** file, which contains accounting information about users logged

in.

Related Information

The ttyslot subroutine.

The **failedlogin**, **utmp**, or **wtmp** file.

getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag, setvfsent, or endvfsent Subroutine

Purpose

Gets a vfs file entry.

Library

Standard C Library(libc.a)

Syntax

```
#include <sys/vfs.h>
#include <sys/vmount.h>
struct vfs_ent *getvfsent()

struct vfs_ent *getvfsbytype( vfsType)
int vfsType;

struct vfs_ent *getvfsbyname( vfsName)
char *vfsName;
```

```
struct vfs_ent *getvfsbyflag( vfsFlag)
int vfsFlag;
void setvfsent( )
void endvfsent( )
```

Description

Attention: All information is contained in a static area and so must be copied to be saved.

The getvfsent subroutine, when first called, returns a pointer to the first vfs_ent structure in the file. On the next call, it returns a pointer to the next vfs ent structure in the file. Successive calls are used to search the entire file.

The vfs_ent structure is defined in the vfs.h file and it contains the following fields:

```
char vfsent name;
int vfsent type;
int vfsent flags;
char *vfsent mnt hlpr;
char *vfsent fs hlpr;
```

The **getvfsbytype** subroutine searches from the beginning of the file until it finds a **vfs** type matching the vfsType parameter. The subroutine then returns a pointer to the structure in which it was found.

The **getyfsbyname** subroutine searches from the beginning of the file until it finds a **yfs** name matching the vfsName parameter. The search is made using flattened names; the search-string uses ASCII equivalent characters.

The getvfsbytype subroutine searches from the beginning of the file until it finds a type matching the *vfsType* parameter.

The **getvfsbyflag** subroutine searches from the beginning of the file until it finds the entry whose flag corresponds flags defined in the vfs.h file. Currently, these are VFS DFLT LOCAL and VFS_DFLT_REMOTE.

The **setvfsent** subroutine rewinds the **vfs** file to allow repeated searches.

The **endvfsent** subroutine closes the **vfs** file when processing is complete.

Parameters

vfsType Specifies a vfs type. vfsName. Specifies a vfs name.

Specifies either VFS_DFLT_LOCAL or VFS_DFLT_REMOTE. vfsFlag

Return Values

The getvfsent, getvfsbytype, getvfsbyname, and getvfsbyflag subroutines return a pointer to a vfs ent structure containing the broken-out fields of a line in the /etc/vfs file. If an end-of-file character or an error is encountered on reading, a null pointer is returned.

Files

/etc/vfs Describes the virtual file system (VFS) installed on the system.

Related Information

The getfsent, getfsspec, getfsfile, getfstype, setfsent, or endfsent ("getfsent, getfsspec, getfsfile, getfstype, setfsent, or endfsent Subroutine" on page 316) subroutine.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

getwc, fgetwc, or getwchar Subroutine

Purpose

Gets a wide character from an input stream.

Library

Standard I/O Package (libc.a)

Syntax

```
#include <stdio.h>
wint t getwc ( Stream)
FILE *Stream;
wint t fgetwc (Stream)
FILE *Stream;
wint t getwchar (void)
```

Description

The **fgetwc** subroutine obtains the next wide character from the input stream specified by the *Stream* parameter, converts it to the corresponding wide character code, and advances the file position indicator the number of bytes corresponding to the obtained multibyte character. The getwc subroutine is equivalent to the fgetwc subroutine, except that when implemented as a macro, it may evaluate the Stream parameter more than once. The getwchar subroutine is equivalent to the getwc subroutine with stdin (the standard input stream).

The first successful run of the fgetc ("getc, getchar, fgetc, or getw Subroutine" on page 300), fgets ("gets or fgets Subroutine" on page 366), fgetwc, fgetws ("getws or fgetws Subroutine" on page 399), fread ("fread or fwrite Subroutine" on page 267), fscanf, getc ("getc, getchar, fgetc, or getw Subroutine" on page 300), getchar ("getc, getchar, fgetc, or getw Subroutine" on page 300), gets ("gets or fgets Subroutine" on page 366), or scanf subroutine using a stream that returns data not supplied by a prior call to the ungetc or **ungetwc** subroutine marks the st atime field for update.

Parameters

Stream Specifies input data.

Return Values

Upon successful completion, the getwc and fgetwc subroutines return the next wide character from the input stream pointed to by the Stream parameter. The getwchar subroutine returns the next wide character from the input stream pointed to by stdin.

If the end of the file is reached, an indicator is set and WEOF is returned. If a read error occurs, an error indicator is set, **WEOF** is returned, and the **errno** global variable is set to indicate the error.

Error Codes

If the **getwc**, **fgetwc**, or **getwchar** subroutine is unsuccessful because the stream is not buffered or data needs to be read into the buffer, it returns one of the following error codes:

EAGAIN Indicates that the O_NONBLOCK flag is set for the file descriptor underlying the

Stream parameter, delaying the process.

EBADF Indicates that the file descriptor underlying the Stream parameter is not valid and

cannot be opened for reading.

EINTR Indicates that the process has received a signal that terminates the read operation. Indicates that a physical error has occurred, or the process is in a background process **EIO**

> group attempting to read from the controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.

EOVERFLOW Indicates that the file is a regular file and an attempt has been made to read at or

beyond the offset maximum associated with the corresponding stream.

The **getwc**, **fgetwc**, or **getwchar** subroutine is also unsuccessful due to the following error conditions:

ENOMEM Indicates that storage space is insufficient.

ENXIO Indicates that the process sent a request to a nonexistent device, or the device cannot

handle the request.

EILSEQ Indicates that the wc wide-character code does not correspond to a valid character.

Related Information

Other wide character I/O subroutines: getws or fgetws ("getws or fgetws Subroutine" on page 399) subroutine, putwc, putwchar, or fputwc ("putwc, putwchar, or fputwc Subroutine" on page 995) subroutine, putws or fputws ("putws or fputws Subroutine" on page 997) subroutine, unqetwc subroutine.

Related standard I/O subroutines: fopen, freopen, or fdopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, gets or fgets ("gets or fgets Subroutine" on page 366) subroutine, fread ("fread or fwrite Subroutine" on page 267) subroutine, fwrite ("fread or fwrite Subroutine" on page 267) subroutine, printf, fprintf, sprintf, wsprintf, vprintf, vprintf, vsprintf, or vwsprintf ("printf, fprintf, sprintf, snprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine, putc, putchar, fputc, or putw ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine, puts or fputs ("puts or fputs Subroutine" on page 994) subroutine.

Subroutines, Example Programs, and Libraries and Understanding Wide Character Input/Output Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

getwd Subroutine

Purpose

Gets current directory path name.

Library

Standard C Library (libc.a)

Syntax

```
#include <unistd.h>
char *getwd ( PathName)
char *PathName;
```

Description

The getwd subroutine determines the absolute path name of the current directory, then copies that path name into the area pointed to by the PathName parameter.

The maximum path-name length, in characters, is set by the PATH_MAX value, as specified in the limits.h file.

Parameters

PathName

Points to the full path name.

Return Values

If the call to the getwd subroutine is successful, a pointer to the absolute path name of the current directory is returned. If an error occurs, the getwd subroutine returns a null value and places an error message in the *PathName* parameter.

Related Information

The getcwd ("getcwd Subroutine" on page 308) subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

getws or fgetws Subroutine

Purpose

Gets a string from a stream.

Library

Standard I/O Library (libc.a)

Syntax

```
#include <stdio.h>
wchar_t *fgetws ( WString, Number, Stream)
wchar t *WString;
int Number;
FILE *Stream;
wchar_t *getws (WString)
wchar t *WString;
```

Description

The fgetws subroutine reads characters from the input stream, converts them to the corresponding wide character codes, and places them in the array pointed to by the WString parameter. The subroutine continues until either the number of characters specified by the Number parameter minus 1 are read or the subroutine encounters a new-line or end-of-file character. The fgetws subroutine terminates the wide character string specified by the WString parameter with a null wide character.

The getws subroutine reads wide characters from the input stream pointed to by the standard input stream (stdin) into the array pointed to by the WString parameter. The subroutine continues until it encounters a new-line or the end-of-file character, then it discards any new-line character and places a null wide character after the last character read into the array.

Parameters

WStrina Points to a string to receive characters. Points to the FILE structure of an open file. Stream

Number Specifies the maximum number of characters to read.

Return Values

If the **getws** or **fgetws** subroutine reaches the end of the file without reading any characters, it transfers no characters to the String parameter and returns a null pointer. If a read error occurs, the getws or fgetws subroutine returns a null pointer and sets the errno global variable to indicate the error.

Error Codes

If the **getws** or **fgetws** subroutine is unsuccessful because the stream is not buffered or data needs to be read into the stream's buffer, it returns one or more of the following error codes:

EAGAIN Indicates that the **O_NONBLOCK** flag is set for the file descriptor underlying the *Stream*

parameter, and the process is delayed in the fgetws subroutine.

EBADF Indicates that the file descriptor specifying the Stream parameter is not a read-access file. **EINTR** Indicates that the read operation is terminated due to the receipt of a signal, and either no

data was transferred or the implementation does not report partial transfer for this file.

EIO Indicates that insufficient storage space is available. ENOMEM Indicates that insufficient storage space is available.

EILSEQ Indicates that the data read from the input stream does not form a valid character.

Related Information

Other wide character I/O subroutines: fgetwc ("getwc, fgetwc, or getwchar Subroutine" on page 397) subroutine, fputwc ("putwc, putwchar, or fputwc Subroutine" on page 995) subroutine, fputws ("putws or fputws Subroutine" on page 997) subroutine, **getwc** ("getwc, fgetwc, or getwchar Subroutine" on page 397) subroutine, getwchar ("getwc, fgetwc, or getwchar Subroutine" on page 397) subroutine, putwc ("putwc, putwchar, or fputwc Subroutine" on page 995) subroutine, putwchar ("putwc, putwchar, or fputwc Subroutine" on page 995) subroutine, putws ("putws or fputws Subroutine" on page 997) subroutine. ungetwc subroutine.

Related standard I/O subroutines: fdopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, fgetc ("getc, getchar, fgetc, or getw Subroutine" on page 300) subroutine, fgets ("gets or fgets Subroutine" on page 366) subroutine, fopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, fprintf, ("printf, sprintf, sprintf, sprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine, fputc ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine, fputs ("puts or fputs Subroutine" on page 994) subroutine, fread ("fread or fwrite Subroutine" on page 267) subroutine, freopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, **fscanf** subroutine, **fwrite** ("fread or fwrite Subroutine" on page 267) subroutine, getc ("getc, getchar, fgetc, or getw Subroutine" on page 300) subroutine, getchar ("getc, getchar, fgetc, or getw Subroutine" on page 300) subroutine, gets ("gets or fgets Subroutine" on page 366) subroutine, printf ("printf, fprintf, sprintf, sprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine, putc ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine, putchar ("putc,

putchar, fputc, or putw Subroutine" on page 990) subroutine, puts ("puts or fputs Subroutine" on page 994) subroutine, putw ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine, scanf subroutine, sprintf ("printf, fprintf, sprintf, sprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine, ungetc subroutine.

Understanding Wide Character Input/Output Subroutines and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

alob Subroutine

Purpose

Generates path names.

Library

Standard C Library (libc.a)

Syntax

```
#include <glob.h>
int glob (Pattern, Flags, (Errfunc)(), Pglob)
const char *Pattern;
int Flags;
int *Errfunc (Epath, Eerrno)
const char *Epath;
int Eerrno;
glob t *Pglob;
```

Description

The **glob** subroutine constructs a list of accessible files that match the *Pattern* parameter.

The **glob** subroutine matches all accessible path names against this pattern and develops a list of all matching path names. To have access to a path name, the **glob** subroutine requires search permission on every component of a path except the last, and read permission on each directory of any file name component of the Pattern parameter that contains any of the special characters * (asterisk), ? (question mark), or [(left bracket). The **glob** subroutine stores the number of matched path names and a pointer to a list of pointers to path names in the Pglob parameter. The path names are in sort order, based on the setting of the LC COLLATE category in the current locale. The first pointer after the last path name is a null character. If the pattern does not match any path names, the returned number of matched paths is zero.

Parameters

Pattern

Contains the file name pattern to compare against accessible path names.

Flags Controls the customizable behavior of the **glob** subroutine.

> The *Flags* parameter controls the behavior of the **glob** subroutine. The *Flags* value is the bitwise inclusive OR of any of the following constants, which are defined in the glob.h file:

GLOB APPEND

Appends path names located with this call to any path names previously located. If the

GLOB APPEND constant is not set, new path names overwrite previous entries in the Pglob array. The GLOB APPEND constant should not be set on the first call to the glob subroutine. It may, however, be set on subsequent calls.

The GLOB_APPEND flag can be used to append a new set of path names to those found in a previous call to the glob subroutine. If the GLOB_APPEND flag is specified in the Flags parameter, the following rules apply:

- If the application sets the GLOB DOOFFS flag in the first call to the glob subroutine, it is also set in the second. The value of the Pglob parameter is not modified between the calls.
- If the application did not set the GLOB_DOOFFS flag in the first call to the glob subroutine, it is not set in the second.
- After the second call, the Palob parameter points to a list containing the following:
 - Zero or more null characters, as specified by the GLOB_DOOFFS flag.
 - Pointers to the path names that were in the Pglob list before the call, in the same order as after the first call to the glob subroutine.
 - Pointers to the new path names generated by the second call, in the specified order.
- · The count returned in the Pglob parameter is the total number of path names from the two calls.
- The application should not modify the *Pglob* parameter between the two calls.

It is the caller's responsibility to create the structure pointed to by the *Pglob* parameter. The **glob** subroutine allocates other space as needed.

GLOB DOOFFS

Uses the gl_offs structure to specify the number of null pointers to add to the beginning of the **gl pathv** component of the *Pglob* parameter.

GLOB ERR

Causes the **glob** subroutine to return when it encounters a directory that it cannot open or read. If the GLOB_ERR flag is not set, the glob subroutine continues to find matches if it encounters a directory that it cannot open or read.

GLOB MARK

Specifies that each path name that is a directory should have a / (slash) appended.

GLOB NOCHECK

If the Pattern parameter does not match any path name, the glob subroutine returns a list consisting only of the Pattern parameter, and the number of matched patterns is one.

GLOB NOSORT

Specifies that the list of path names need not be sorted. If the GLOB_NOSORT flag is not set, path names are collated according to the current locale.

GLOB QUOTE

If the GLOB_QUOTE flag is set, a \ (backslash) can be used to escape metacharacters.

Errfunc

Specifies an optional subroutine that, if specified, is called when the glob subroutine detects an error condition.

Pglob Contains a pointer to a **glob t** structure. The structure is allocated by the caller. The array of structures containing the file names matching the Pattern parameter are defined by the glob subroutine. The last entry is a null pointer.

Epath Specifies the path that failed because a directory could not be opened or read.

Eerrno Specifies the errno value of the failure indicated by the Epath parameter. This value is set by the opendir, readdir, or stat subroutines.

Return Values

On successful completion, the **glob** subroutine returns a value of 0. The *Pglob* parameter returns the number of matched path names and a pointer to a null-terminated list of matched and sorted path names. If the number of matched path names in the Pglob parameter is zero, the pointer in the Pglob parameter is undefined.

Error Codes

If the **glob** subroutine terminates due to an error, it returns one of the nonzero constants below. These are defined in the glob.h file. In this case, the Pglob values are still set as defined in the Return Values section.

GLOB ABORTED Indicates the scan was stopped because the GLOB ERROR flag was set or the

subroutine specified by the **errfunc** parameter returned a nonzero value.

GLOB_NOSPACE Indicates a failed attempt to allocate memory.

If, during the search, a directory is encountered that cannot be opened or read and the Errfunc parameter is not a null value, the **glob** subroutine calls the subroutine specified by the **errfunc** parameter with two arguments:

- The Epath parameter specifies the path that failed.
- The Eerrno parameter specifies the value of the errno global variable from the failure, as set by the opendir, readdir, or stat subroutine.

If the subroutine specified by the Errfunc parameter is called and returns nonzero, or if the GLOB ERR flag is set in the *Flags* parameter, the **glob** subroutine stops the scan and returns **GLOB ABORTED** after setting the Pglob parameter to reflect the paths already scanned. If GLOB ERR is not set and either the *Errfunc* parameter is null or *errfunc returns zero, the error is ignored.

The *Pglob* parameter has meaning even if the **glob** subroutine fails. Therefore, the **glob** subroutine can report partial results in the event of an error. However, if the number of matched path names is 0, the pointer in the *Pglob* parameter is unspecified even if the **glob** subroutine did not return an error.

Examples

The GLOB_NOCHECK flag can be used with an application to expand any path name using wildcard characters. However, the GLOB NOCHECK flag treats the pattern as just a string by default. The sh command can use this facility for option parameters, for example.

The GLOB_DOOFFS flag can be used by applications that build an argument list for use with the execv, execve, or execvp subroutine. For example, an application needs to do the equivalent of 1s -1 *.c, but for some reason cannot. The application could still obtain approximately the same result using the sequence:

```
globbuf.gl offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
globbuf.gl pathv[0] = "ls";
globbuf.gl_pathv[1] ="-l":
execvp ("ls", &globbuf.gl pathv[0]);
```

Using the same example, 1s -1 *.c *.h could be approximated using the GLOB APPEND flag as follows:

```
globbuf.gl offs = 2;
glob ("*.c", GLOB DOOFFS, NULL, &globbuf);
glob ("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
```

The new path names generated by a subsequent call with the **GLOB_APPEND** flag set are not sorted together with the previous path names. This is the same way the shell handles path name expansion when multiple expansions are done on a command line.

Related Information

The exec: execl, execv, execle, execve, execlp, execvp, or exect ("exec: execl, execle, execlp, execv, execve, execve, execvp, or exect Subroutine" on page 196) subroutine, fnmatch ("fnmatch Subroutine" on page 243) subroutine, opendir, readdir, telldir, seekdir, rewinddir, or closedir ("opendir, readdir, telldir, seekdir, rewinddir, or closedir Subroutine" on page 771) subroutine, statx, stat, Istat, fstatx, fstat, fullstat, or ffullstat subroutine.

The Is command.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

globfree Subroutine

Purpose

Frees all memory associated with the pglob parameter.

Library

Standard C Library (libc.a)

Syntax

#include <glob.h>
void globfree (pglob)
glob_t *pglob;

Description

The **globfree** subroutine frees any memory associated with the *pglob* parameter due to a previous call to the **glob** subroutine.

Parameters

pglob

Structure containing the results of a previous call to the glob subroutine.

Related Information

The **glob** ("glob Subroutine" on page 401) subroutine.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

grantpt Subroutine

Purpose

Changes the mode and ownership of a pseudo-terminal device.

Library

Standard C Library (libc.a)

Syntax

#include <stdlib.h>

int grantpt (FileDescriptor) int FileDescriptor;

Description

The grantpt subroutine changes the mode and the ownership of the slave pseudo-terminal associated with the master pseudo-terminal device defined by the FileDescriptor parameter. The user ID of the slave pseudo-terminal is set to the real UID of the calling process. The group ID of the slave pseudo-terminal is set to an unspecified group ID. The permission mode of the slave pseudo-terminal is set to readable and writeable by the owner, and writeable by the group.

Parameters

FileDescriptor

Specifies the file descriptor of the master pseudo-terminal device.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The grantpt function may fail if:

EBADF The fildes argument is not a valid open file descriptor.

EINVAL The fildes argument is not associated with a master pseudo-terminal device. **EACCES** The corresponding slave pseudo-terminal device could not be accessed.

Related Information

The unlockpt subroutine.

The Input and Output Handling Programmer's Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

HBA_CloseAdapter Subroutine

Purpose

Closes the adapter opened by the HBA_OpenAdapter subroutine.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

void HBA CloseAdapter (handle) HBA_HANDLE handle;

Description

The **HBA_CloseAdapter** subroutine closes the file associated with the file handle that was the result of a call to the **HBA_OpenAdapter** subroutine. The **HBA_CloseAdapter** subroutine calls the **close** subroutine, and applies it to the file handle. After performing the operation, the handle is set to NULL.

Parameters

handle Specifies the open file descriptor obtained from a successful call to the **open** subroutine.

Related Information

The "HBA_OpenAdapter Subroutine" on page 416.

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA_FreeLibrary Subroutine

Purpose

Frees all the resources allocated to build the Common HBA API Library.

Library

Common Host Bus Adapter Library (IibHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

HBA_STATUS HBA_FreeLibrary ()

Description

The **HBA_FreeLibrary** subroutine frees all resources allocated to build the Common HBA API library. This subroutine must be called after calling any other routine from the Common HBA API library.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

HBA_STATUS_OK

A value of 0 on successful completion.

HBA STATUS ERROR

A value of 1 if an error occurred.

Related Information

The "HBA_LoadLibrary Subroutine" on page 415.

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA_GetAdapterAttributes, HBA_GetPortAttributes, HBA_GetDiscoveredPortAttributes, HBA_GetPortAttributesByWWN Subroutine

Purpose

Gets the attributes of the end device's adapter, port, or remote port.

Library

Common Host Bus Adapter Library (IibHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

```
HBA_STATUS HBA_GetAdapterAttributes (handle, hbaattributes)
HBA_STATUS HBA_GetAdapterPortAttributes (handle, portindex, portattributes)
HBA_STATUS HBA_GetDiscoveredPortAttributes (handle, portindex, discoveredportindex, portattributes)
HBA_STATUS HBA_GetPortAttributesByWWN (handle, PortWWN, portattributes)

HBA_HANDLE handle;
HBA_ADAPTERATTRIBUTES *hbaattributes;
HBA_UINT32 portindex;
HBA_PORTATTRIBUTES *portattributes;
HBA_UINT32 discoveredportindex;
```

Description

HBA WWN PortWWN;

The **HBA_GetAdapterAttributes** subroutine queries the ODM and makes system calls to gather information pertaining to the adapter. This information is returned to the **HBA_ADAPTERATTRIBUTES** structure. This structure is defined in the **/usr/include/sys/hbaapi.h** file.

The HBA_GetAdapterAttributes, HBA_GetAdapterPortAttributes, HBA_GetDiscoveredPortAttributes, and HBA_GetPortAttributesByWWN subroutines return the attributes of the adapter, port or remote port.

These attributes include:

- Manufacturer
- SerialNumber
- Model
- ModelDescription
- NodeWWN
- NodeSymbolicName
- HardwareVersion
- DriverVersion
- OptionROMVersion
- FirmwareVersion
- VendorSpecificID
- NumberOfPorts
- Drivername

The HBA_GetAdapterPortAttributes, HBA_GetDiscoveredPortAttributes, and HBA GetPortAttributesByWWN subroutines also query the ODM and make system calls to gather

information. The gathered information pertains to the port attached to the adapter or discovered on the network. The attributes are stored in the HBA PORTATTRIBUTES structure. This structure is defined in the /usr/include/sys/hbaapi.h file.

These attributes include:

- NodeWWN
- PortWWN
- PortFcld
- PortType
- PortState
- PortSupportedClassofService
- PortSupportedFc4Types
- PortActiveFc4Types
- OSDeviceName
- PortSpeed
- NumberofDiscoveredPorts
- PortSymbolicName
- PortSupportedSpeed
- PortMaxFrameSize
- FabricName

The HBA GetAdapterPortAttributes subroutine returns the attributes of the attached port.

The HBA_GetDiscoveredPortAttributes, and HBA_GetPortAttributesByWWN subroutines return the same information. However, these subroutines differ in the way they are called, and in the way they acquire the information.

Parameters

handle Specifies the open file descriptor obtained from a successful call to the open

subroutine.

Points to an HBA_AdapterAttributes structure, which is used to store information hbaatributes

pertaining to the Host Bus Adapter.

portindex Specifies the index number of the port where the information was obtained.

Points to an HBA PortAttributes structure used to store information pertaining to the portattributes

port attached to the Host Bus Adapter.

discoveredportindex Specifies the index of the attached port discovered over the network. **PortWWN** Specifies the world wide name or port name of the target device.

Return Values

Upon successful completion, the attributes and a value of HBA STATUS OK, or 0 are returned.

If no information for a particular attribute is available, a null value is returned for that attribute. HBA_STATUS_ERROR or 1 is returned if certain ODM queries or system calls fail while trying to retrieve the attributes.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

HBA_STATUS_OK

A value of 0 on successful completion.

HBA STATUS ERROR HBA_STATUS_ERROR_INVALID_HANDLE

HBA STATUS ERROR ARG HBA STATUS ERROR ILLEGAL WWN A value of 1 if an error occurred. A value of 3 if there was an invalid file

handle.

A value of 4 if there was a bad argument. A value of 5 if the world wide name was not

recognized.

Related Information

"HBA_GetAdapterName Subroutine," and "HBA_GetNumberOfAdapters Subroutine" on page 412.

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA_GetAdapterName Subroutine

Purpose

Gets the name of a Common Host Bus Adapter.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

HBA STATUS HBA GetAdapterName (adapterindex, adaptername) **HBA UINT32** adapterindex; char *adaptername;

Description

The HBA_GetAdapterName subroutine gets the name of a Common Host Bus Adapter. The adapterindex parameter is an index into an internal table containing all FCP adapters on the machine. The adapterindex parameter is used to search the table and obtain the adapter name. The name of the adapter is returned in the form of mgfdomain-model-adapterindex. The name of the adapter is used as an argument for the HBA_OpenAdapter subroutine. From the HBA_OpenAdapter subroutine, the file descriptor will be obtained where additional Common HBA API routines can then be called using the file descriptor as the argument.

Parameters

adapterindex Specifies the index of the adapter held in the adapter table for which the name of the adapter

is to be returned.

Points to a character string that will be used to hold the name of the adapter. adaptername

Return Values

Upon successful completion, the HBA_GetAdapterName subroutine returns the name of the adapter and a 0, or a status code of HBA STATUS OK. If unsuccessful, a null value will be returned for adaptername and an value of 1, or a status code of HBA_STATUS_ERROR.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

HBA_STATUS_OK
HBA_STATUS_ERROR
HBA_STATUS_ERROR_NOT_SUPPORTED
HBA_STATUS_ERROR_INVALID_HANDLE

HBA_STATUS_ERROR_ARG
HBA_STATUS_ERROR_ILLEGAL_WWN

HBA_STATUS_ERROR_ILLEGAL_INDEX

HBA_STATUS_ERROR_MORE_DATA
HBA_STATUS_ERROR_STALE_DATA

HBA_STATUS_SCSI_CHECK_CONDITION

HBA_STATUS_ERROR_BUSY

HBA_STATUS_ERROR_TRY_AGAIN

HBA_STATUS_ERROR_UNAVAILABLE

A value of 0 on successful completion.

A value of 1 if an error occurred.

A value of 2 if the function is not supported.

A value of 3 if there was an invalid file handle.

A value of 4 if there was a bad argument. A value of 5 if the world wide name was

not recognized.

A value of 6 if an index was not recognized.

A value of 7 if a larger buffer is required. A value of 8 if information has changed

since the last call to the **HBA RefreshInformation** subroutine.

A value of 9 if a SCSI Check Condition was reported

A value of 10 if the adapter was busy or reserved. Try again later.

A value of 11 if the request timed out. Try again later.

A value of 12 if the referenced HBA has been removed or deactivated.

Related Information

The "HBA_GetNumberOfAdapters Subroutine" on page 412.

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA_GetFcpPersistentBinding Subroutine

Purpose

Gets persistent binding information of SCSI LUNs.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

HBA_STATUS HBA_GetFcpPersistentBinding (handle, binding)
HBA_HANDLE handle;
PHBA_FCPBinding binding;

Description

For the specified HBA_HANDLE, the **HBA_GetFcpPersistentBinding** subroutine returns the full binding information of local SCSI LUNs to FCP LUNs for each child of the specified HBA_HANDLE. Applications must allocate memory for the **HBA_FCPBINDING** structure, and also pass to the subroutine the number of entries allocated. If the subroutine determines that the structure is not large enough to represent the full binding information, it will set the *NumberOfEntries* variable to the correct value and return an error.

Parameters

handle
binding

An HBA_HANDLE to an open adapter.

A pointer to a structure containing the binding information of the handle's children. The HBA_FCPBINDING structure has the following form:

struct HBA_FCPBInding {
 HBA_UINT32 NumberOfEntries;
 HBA_FCPBINDINGENTRY entry[1]; /* Variable length array */

The size of the structure is determined by the calling application, and is passed in by the *NumberOfEntries* variable.

Return Values

Upon successful completion, HBA_STATUS_OK is returned, and the *binding* parameter points to the full binding structure. If the application has not allocated enough space for the full binding, HBA_STATUS_ERROR_MORE_DATA is returned and the *NumberOfEntries* field in the binding structure is set to the correct value.

Error Codes

If there is insufficient space allocated for the full binding. HBA_STATUS_ERROR_MORE_DATA is returned.

Related Information

The "HBA_GetFcpTargetMapping Subroutine."

HBA_GetFcpTargetMapping Subroutine

Purpose

Gets mapping of OS identification to FCP indentification for each child of the specified HBA_HANDLE.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

```
#include <sys/hbaapi.h>
HBA_STATUS HBA_GetFcpTargetMapping (handle, mapping)
HBA_HANDLE handle;
PHBA_FCPTARGETMAPPING mapping;
```

Description

For the specified HBA_HANDLE, the **HBA_GetFcpTargetMapping** subroutine maps OS identification of all its SCSI logical units to their FCP indentification. Applications must allocate memory for the **HBA_FCPTargetMapping** structure, and also pass to the subroutine the number of entries allocated. If the subroutine determines that the structure is not large enough to represent the entire mapping, it will set the *NumberOfEntries* variable to the correct value and return an error.

Parameters

handle An HBA_HANDLE to an open adapter.

mapping

A pointer to a structure containing a mapping of the handle's children. The HBA_FCPTARGETMAPPING structure has the following form:

```
struct HBA FCPTargetMapping (
HBA UINT32 NumberOfEntries;
HBA FCPSCSIENTRY entry[1] /* Variable length array containing mappings */
```

The size of the structure is determined by the calling application, and is passed in by the NumberOfEntries variable.

Return Values

If successful, HBA_STATUS_OK is returned and the mapping parameter points to the full mapping structure. If the application has not allocated enough space for the full mapping, HBA_STATUS_ERROR_MORE_DATA is returned, and the NumberOfEntries field in the mapping structure is set to the correct value.

Error Codes

If there is insufficient space allocated for the full mapping, HBA STATUS ERROR MORE DATA is returned.

Related Information

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA_GetNumberOfAdapters Subroutine

Purpose

Returns the number of adapters discovered on the system.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

```
#include <sys/hbaapi.h>
HBA_UINT32 HBA_GetNumberOfAdapters ()
```

Description

The HBA_GetNumberOfAdapters subroutine returns the number of adapters that support the Common Host Bus Adapter API library. The HBA_GetNumberOfAdapters subroutine queries the ODM for an active FCP adapter present on the machine. The number of adapters is used to build a table to store adapter information. The number of adapters will be used to build an internal table containing all the FCP adapters available on the machine.

Return Values

The HBA_GetNumberOfAdapters subroutine returns an integer representing the number of adapters on the machine.

Related Information

The "HBA_GetAdapterName Subroutine" on page 409.

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA_GetPortStatistics Subroutine

Purpose

Gets the statistics for a Host Bus Adapter (HBA).

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

HBA_STATUS HBA_GetPortStatistics (handle, portindex, portstatistics) HBA HANDLE handle; **HBA UINT32** portindex; HBA_PORTSTATISTICS *portstatistics;

Description

The HBA_GetPortStatistics subroutine retrieves the statistics for the specified adapter. Only single-port adapters are supported, and the *portindex* parameter is disregarded. The exact meaning of events being counted for each statistic is vendor specific. The HBA_PORTSTATISTICS structure includes the following fields:

- SecondsSinceLastReset
- **TxFrames**
- TxWords
- RxFrames
- RxWords
- LIPCount
- NOSCount
- ErrorFrames
- DumpedFrames
- LinkFailureCount
- LossOfSyncCount
- LossOfSignalCount
- PrimitiveSeqProtocolErrCount
- InvalidTxWordCount
- InvalidCRCCount

Parameters

handle HBA_HANDLE to an open adapter.

portindex Not used.

portstatistics Pointer to an HBA PORTSTATISTICS structure.

Return Values

Upon successful completion, HBA_STATUS_OK is returned. If the subroutine is unable to retrieve the statistics for an HBA, it returns HBA STATUS ERROR.

HBA_GetRNIDMgmtInfo Subroutine

Purpose

Sends a SCSI GET RNID command to a remote port of the end device.

Library

Common Host Bus Adapter Library (IibHBAAPI.a)

Syntax

```
#include <sys/hbaapi.h>
HBA STATUS HBA GetRNIDMgmtInfo (handle, pInfo)
HBA HANDLE handle;
HBA MGMTINFO *pInfo;
```

Description

The **HBA SetRNIDMgmtInfo** subroutine sends a **SCSI GET RNID** (Request Node Identification Data) command through a call to ioctl with the SCIOLCHBA operation as its argument. The arg parameter for the SCIOLCHBA operation is the address of a scsi chba structure. This structure is defined in the /usr/include/sys/scsi buf.h file. The scsi chba parameter block allows the caller to select the GET RNID command to be sent to the adapter. The pinfo structure stores the RNID data returned from SCIOLCHBA. The plnfo structure is defined in the /usr/include/sys/hbaapi.h file. The structure includes:

- wwn
- unittype
- PortId
- NumberOfAttachedNodes
- IPVersion
- UDPort
- IPAddress
- reserved
- TopologyDiscoveryFlags

If successful, the GET RNID data in plnfo is returned from the adapter.

Parameters

handle Specifies the open file descriptor obtained from a successful call to the open subroutine. pInfo Specifies the structure containing the information to get or set from the RNID command

Return Values

Upon successful completion, the HBA_GetRNIDMgmtInfo subroutine returns a pointer to a structure containing the data from the GET RNID command and a value of HBA STATUS OK, or a value of 0. If unsuccessful, a null value is returned along with a value of HBA STATUS ERROR, or a value of 1.

Upon successful completion, the HBA_SetRNIDMgmtInfo subroutine returns a value of HBA_STATUS_OK, or a value of 0. If unsuccessful, an HBA_STATUS_ERROR value, or a value of 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

HBA STATUS OK HBA_STATUS_ERROR HBA STATUS ERROR INVALID HANDLE

A value of 0 on successful completion. A value of 1 if an error occurred. A value of 3 if there was an invalid file handle.

Related Information

"HBA_SendScsiInquiry Subroutine" on page 422, "HBA_SendReadCapacity Subroutine" on page 418, "HBA_SendCTPassThru Subroutine" on page 417, "HBA_SendReportLUNs Subroutine" on page 419, "HBA SendRNID Subroutine" on page 421, and "HBA SetRNIDMgmtInfo Subroutine" on page 423.

SCSI Adapter Device Driver in AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 2.

Special Files in AIX 5L Version 5.2 Files Reference.

SCSI Subsystem Overview, A Typical Initiator-Mode SCSI Driver Transaction Sequence, Required SCSI Adapter Device Driver ioctl Commands, Understanding the Execution of Initiator I/O Requests, SCSI Error Recovery, and Understanding the sc_buf Structure in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

HBA GetVersion Subroutine

Purpose

Returns the version number of the Common HBA API.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

HBA UINT32 HBA GetVersion ()

Description

The HBA GetVersion subroutine returns the version number representing the release of the Common HBA API.

Return Values

Upon successful completion, the HBA GetVersion subroutine returns an integer value designating the version number of the Common HBA API.

Related Information

"HBA_LoadLibrary Subroutine" and "HBA_FreeLibrary Subroutine" on page 406.

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA LoadLibrary Subroutine

Purpose

Loads a vendor specific library from the Common HBA API.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

HBA STATUS HBA LoadLibrary ()

Description

The **HBA_LoadLibrary** subroutine loads a vendor specific library from the Common HBA API. This library must be called first before calling any other routine from the Common HBA API.

Return Values

The HBA_LoadLibrary subroutine returns a value of 0, or HBA_STATUS_OK.

Related Information

The "HBA FreeLibrary Subroutine" on page 406.

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA_OpenAdapter Subroutine

Purpose

Opens the specified adapter for reading.

Library

Common Host Bus Adapter Library (IibHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

HBA_HANDLE HBA_OpenAdapter (adaptername)
char *adaptername;

Description

The **HBA_OpenAdapter** subroutine opens the adapter for reading for the purpose of getting it ready for additional calls from other subroutines in the Common HBA API.

The **HBA_OpenAdapter** subroutine allows an application to open a specified HBA device, giving the application access to the device through the HBA_HANDLE return value. The library ensures that all access to this HBA_HANDLE between **HBA_OpenAdapter** and **HBA_CloseAdapter** calls is to the same device.

Parameters

adaptername

Specifies a string that contains the description of the adapter as returned by the **HBA_GetAdapterName** subroutine.

Return Values

If successful, the HBA_OpenAdapter subroutine returns an HBA_HANDLE with a value greater than 0. If unsuccessful, the subroutine returns a 0.

Related Information

"HBA_CloseAdapter Subroutine" on page 405, and "HBA_GetAdapterName Subroutine" on page 409.

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA RefreshInformation Subroutine

Purpose

Refreshes stale information from the Host Bus Adapter.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

void HBA RefreshInformation (handle) HBA_HANDLE handle;

Description

The HBA_RefreshInformation subroutine refreshes stale information from the Host Bus Adapter. This would reflect changes to information obtained from calls to the HBA_GetAdapterPortAttributes, or HBA GetDiscoveredPortAttributes subroutine. Once the application calls the HBA RefreshInformation subroutine, it can proceed to the attributes's call to get the new data.

Parameters

handle

Specifies the open file descriptor obtained from a successful call to the open subroutine for which the refresh operation is to be performed.

Related Information

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA SendCTPassThru Subroutine

Purpose

Sends a CT pass through frame.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

HBA_STATUS HBA_SendCTPassThru (handle, pReqBuffer, ReqBufferSize, pRspBuffer, RspBufferSize)
HBA_HANDLE handle;
void *pReqBuffer;
HBA_UINT32 ReqBufferSize;
void *pRspBuffer;
HBA_UINT32 RspBufferSize;

Description

The **HBA_SendCTPassThru** subroutine sends a CT pass through frame to a fabric connected to the specified handle. The CT frame is routed in the fabric according to the *GS TYPE* field in the CT frame.

Parameters

handle HBA_HANDLE to an open adapter.

pReqBuffer Pointer to a buffer that contains the CT request.

RegBufferSize Size of the request buffer.

pRspBuffer Pointer to a buffer that receives the response of the command.

RspBufferSize Size of the response buffer.

Return Values

If successful, HBA STATUS OK is returned, and the pRspBuffer parameter points to the CT response.

Error Codes

If the adapter specified by the *handle* parameter is connected to an arbitrated loop, the **HBA_SendCTPassThru** subroutine returns HBA_STATUS_ERROR_NOT_SUPPORTED. This subroutine is only valid when connected to a fabric.

Related Information

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA_SendReadCapacity Subroutine

Purpose

Sends a SCSI READ CAPACITY command to a Fibre Channel port.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

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

```
HBA_STATUS HBA_SendReadCapacity (handle, portWWN, fcLUN, pRspBuffer, RspBufferSize, pSenseBuffer, SenseBufferSize)
HBA_HANDLE handle;
HBA_WWN portWWN;
HBA_UINT64 fcLUN;
```

void *pRspBuffer; **HBA UINT32** RspBufferSize; void *pSenseBuffer; HBA_UINT32 SenseBufferSize;

Description

The HBA SendReadCapacity subroutine sends a SCSI READ CAPACITY command to the Fibre Channel port connected to the handle parameter and specified by the portWWN and fcLUN parameters.

Parameters

handle HBA HANDLE to an open adapter. portWWN Port world-wide name of an adapter.

fcLUN Fibre Channel LUN to send the SCSI READ CAPACITY command to. Pointer to a buffer that receives the response of the command. pRspBuffer

RspBufferSize Size of the response buffer.

Pointer to a buffer that receives sense information. pSenseBuffer

SenseBufferSize Size of the sense buffer.

Return Values

If successful, HBA_STATUS_OK is returned and the pRspBuffer parameter points to the response to the **READ CAPACITY** command. If an error occurs, HBA_STATUS_ERROR is returned.

Error Codes

If the portWWN value is not a valid world-wide name connected to the specified handle, HBA_STATUS_ERROR_ILLEGAL_WWN is returned. On any other types of failures, HBA STATUS ERROR is returned.

Related Information

The "HBA_SendScsiInquiry Subroutine" on page 422.

Special Files in AIX 5L Version 5.2 Files Reference describes specific qualities of the files that define devices.

HBA_SendReportLUNs Subroutine

Purpose

Sends a SCSI REPORT LUNs command to a remote port of the end device.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

```
HBA_STATUS HBA_SendReportLUNs (handle, PortWWN, pRspBuffer, RspBufferSize, pSenseBuffer, SenseBufferSize)
HBA HANDLE handle;
HBA WWN PortWWN;
void *pRspBuffer;
HBA_UINT32 RspBufferSize;
void *pSenseBuffer;
HBA_UINT32 SenseBufferSize;
```

Description

The HBA SendReportLUNs subroutine sends a SCSI REPORT LUNs command through a call to ioctl with the SCIOLCMD operation as its argument. The arg parameter for the SCIOLCMD operation is the address of a scsi_iocmd structure. This structure is defined in the /usr/include/sys/scsi_buf.h file. The scsi iocmd parameter block allows the caller to select the SCSI and LUN IDs to be gueried. The caller also specifies the SCSI command descriptor block area, command length (SCSI command block length), the time-out value for the command, and a flags field.

If successful, the report LUNs data is returned in pRspBuffer. The returned report LUNs data must be examined to see if the requested LUN exists.

Parameters

handle Specifies the open file descriptor obtained from a successful call to the open subroutine.

PortWWN Specifies the world wide name or port name of the target device.

Points to a buffer containing the requested instruction for a send/read capacity request to an pRspBuffer

open adapter.

RspBufferSize Specifies the size of the buffer to the *pRspBuffer* parameter.

pSenseBuffer Points to a buffer containing the data returned from a send/read capacity request to an open

adapter.

Specifies the size of the buffer to the *pSenseBuffer* parameter. SenseBufferSize 5 6 1

Return Values

Upon successful completion, the HBA SendReportLUNs subroutine returns a buffer in bytes containing the SCSI report of LUNs, a buffer containing the SCSI sense data, and a value of HBA STATUS OK, or a value of 0.

If unsuccessful, an empty buffer for the SCSI report of LUNs, a response buffer containing the failure, and a value of HBA STATUS ERROR, or a value of 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

HBA_STATUS_OK A value of 0 on successful completion. HBA_STATUS_ERROR A value of 1 if an error occurred.

HBA_STATUS_ERROR_INVALID_HANDLE A value of 3 if there was an invalid file handle.

HBA_STATUS_ERROR_ILLEGAL_WWN A value of 5 if the world wide name was not recognized. HBA STATUS SCSI CHECK CONDITION A value of 9 if a SCSI Check Condition was reported.

Related Information

"HBA_SendScsiInquiry Subroutine" on page 422, "HBA_SendReadCapacity Subroutine" on page 418,

"HBA_SendCTPassThru Subroutine" on page 417, "HBA_SendRNID Subroutine" on page 421,

"HBA SetRNIDMgmtInfo Subroutine" on page 423, and "HBA GetRNIDMgmtInfo Subroutine" on page 414.

SCSI Adapter Device Driver in AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 2.

Special Files in AIX 5L Version 5.2 Files Reference.

SCSI Subsystem Overview, A Typical Initiator-Mode SCSI Driver Transaction Sequence, Required SCSI Adapter Device Driver ioctl Commands, Understanding the Execution of Initiator I/O Requests, SCSI Error Recovery, and Understanding the sc buf Structure in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

HBA SendRNID Subroutine

Purpose

Sends an RNID command through a call to SCIOLPAYLD to a remote port of the end device.

Library

Common Host Bus Adapter Library (libHBAAPI.a)

Syntax

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

```
HBA_STATUS HBA_SendRNID (handle, wwn, wwntype, pRspBuffer, RspBufferSize)
HBA HANDLE handle;
HBA WWN wwn;
HBA WWNTYPE wwntype;
void *pRspBuffer;
HBA UINT32 RspBufferSize;
```

Description

The HBA SendRNID subroutine sends a SCSI RNID command with the Node Identification Data Format set to indicate the default Topology Discovery format. This is done through a call to ioctl with the SCIOLPAYLD operation as its argument. The arg parameter for the SCIOLPAYLD operation is the address of an scsi_trans_payld structure. This structure is defined in the /usr/include/sys/scsi_buf.h file. The scsi trans payld parameter block allows the caller to select the SCSI and LUN IDs to be gueried. In addition, the caller must specify the fcph_rnid_payld_t structure to hold the command and the topology format for SCIOLPAYLD. The structure for the fcph_rnid_payld_t structure is defined in the /usr/include/sys/fcph.h file.

If successful, the RNID data is returned in pRspBuffer. The returned RNID data must be examined to see if the requested information exists.

Parameters

Specifies the open file descriptor obtained from a successful call to the open subroutine. handle

Specifies the world wide name or port name of the target device. wwn

wwntype Specifies the type of the world wide name or port name of the target device.

Points to a buffer containing the requested instruction for a send/read capacity request to an pRspBuffer

open adapter.

Specifies the size of the buffer to the *pRspBuffer* parameter. RspBufferSize

Return Values

Upon successful completion, the HBA SendRNID subroutine returns a buffer in bytes containing the SCSI RNID data and a value of HBA STATUS OK, or a value of 0. If unsuccessful, an empty buffer for the SCSI RNID and a value of HBA STATUS ERROR, or a value of 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

HBA STATUS OK A value of 0 on successful completion. **HBA STATUS ERROR** A value of 1 if an error occurred.

HBA_STATUS_ERROR_NOT_SUPPORTED A value of 2 if the function is not supported. HBA_STATUS_ERROR_INVALID_HANDLE A value of 3 if there was an invalid file handle.

HBA STATUS ERROR ILLEGAL WWN A value of 5 if the world wide name was not recognized.

Related Information

"HBA_SendScsiInquiry Subroutine," "HBA_SendReadCapacity Subroutine" on page 418, "HBA_SendCTPassThru Subroutine" on page 417, "HBA_SendReportLUNs Subroutine" on page 419, "HBA_SetRNIDMgmtInfo Subroutine" on page 423, and "HBA_GetRNIDMgmtInfo Subroutine" on page 414.

SCSI Adapter Device Driver in AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 2.

Special Files in AIX 5L Version 5.2 Files Reference.

SCSI Subsystem Overview, A Typical Initiator-Mode SCSI Driver Transaction Sequence, Required SCSI Adapter Device Driver ioctl Commands, Understanding the Execution of Initiator I/O Requests, SCSI Error Recovery, and Understanding the sc_buf Structure in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

HBA_SendScsiInquiry Subroutine

Purpose

Sends a SCSI device inquiry command to a remote port of the end device.

Library

Common Host Bus Adapter Library (IibHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

HBA_STATUS HBA_SendScsiInquiry (handle, PortWWN, fcLUN, EVPD, PageCode, pRspBuffer, RspBufferSize, pSenseBuffer, SenseBufferSize)
HBA_HANDLE handle;
HBA_WNN PortWWN;
HBA_UINT64 fcLUN;
HBA_UINT8 EVPD;
HBA_UINT32 PageCode;
void *pRspBuffer;
HBA_UINT32 RspBufferSize;
void *pSenseBuffer;
HBA_UINT32 SenseBufferSize;

Description

The HBA_SendScsiInquiry subroutine sends a SCSI INQUIRY command through a call to ioctI with the SCIOLINQU operation as its argument. The arg parameter for the SCIOLINQU operation is the address of an scsi_inquiry structure. This structure is defined in the /usr/include/sys/scsi_buf.h file. The scsi_inquiry parameter block allows the caller to select the SCSI and LUN IDs to be queried. If successful, the inquiry data is returned in the pRspBuffer parameter. Successful completion occurs if a device responds at the requested SCSI ID, but the returned inquiry data must be examined to see if the requested LUN exists.

Parameters

handle Specifies the open file descriptor obtained from a successful call to the **open** subroutine.

PortWWN Specifies the world wide name or port name of the target device.

fcLUN Specifies the fcLUN.

EVPD Specifies the value for the EVPD bit. If the value is 1, the Vital Product Data page code

will be specified by the PageCode parameter.

PageCode 1 Specifies the Vital Product Data that is to be requested if the EVPD parameter is set to

pRspBuffer Points to a buffer containing the requested instruction for a send/read capacity request to

an open adapter.

RspBufferSize Specifies the size of the buffer to the *pRspBuffer* parameter.

pSenseBuffer Points to a buffer containing the data returned from a send/read capacity request to an

open adapter.

SenseBufferSize Specifies the size of the buffer to the *pSenseBuffer* parameter.

Return Values

Upon successful completion, the HBA_SendScsilnquiry subroutine returns a buffer in bytes containing the SCSI inquiry, a buffer containing the SCSI sense data, and a value of HBA STATUS OK, or a value of

If unsuccessful, an empty buffer for the SCSI inquiry, a response buffer containing the failure, and a value of HBA_STATUS_ERROR, or a value of 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

HBA STATUS OK A value of 0 on successful completion. HBA_STATUS_ERROR A value of 1 if an error occurred.

HBA STATUS ERROR INVALID HANDLE A value of 3 if there was an invalid file handle. HBA_STATUS_ERROR_ARG A value of 4 if there was a bad argument.

HBA_STATUS_ERROR_ILLEGAL_WWN A value of 5 if the world wide name was not recognized. HBA STATUS SCSI CHECK CONDITION A value of 9 if a SCSI Check Condition was reported.

Related Information

"HBA_SendReportLUNs Subroutine" on page 419, "HBA_SendReadCapacity Subroutine" on page 418,

"HBA_SendCTPassThru Subroutine" on page 417, "HBA_SendRNID Subroutine" on page 421,

"HBA_SetRNIDMgmtInfo Subroutine," and "HBA_GetRNIDMgmtInfo Subroutine" on page 414.

SCSI Adapter Device Driver in AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 2.

Special Files in AIX 5L Version 5.2 Files Reference.

SCSI Subsystem Overview, A Typical Initiator-Mode SCSI Driver Transaction Sequence, Required SCSI Adapter Device Driver ioctl Commands, Understanding the Execution of Initiator I/O Requests, SCSI Error Recovery, and Understanding the sc buf Structure in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

HBA_SetRNIDMgmtInfo Subroutine

Purpose

Sends a SCSI SET RNID command to a remote port of the end device.

Library

Common Host Bus Adapter Library (IibHBAAPI.a)

Syntax

#include <sys/hbaapi.h>

HBA_STATUS HBA_SetRNIDMgmtInfo (handle, pInfo)
HBA_HANDLE handle;
HBA_MGMTINFO *pInfo;

Description

The **HBA_SetRNIDMgmtInfo** subroutine sends a **SCSI SET RNID** (Request Node Identification Data) command with the **SCIOLCHBA** operation as its argument. This is done through a call to **ioctl**. The *arg* parameter for the **SCIOLCHBA** operation is the address of a **scsi_chba** structure. This structure is defined in the **/usr/include/sys/scsi_buf.h** file. The *scsi_chba* parameter block allows the caller to select the **SET RNID** command to be sent to the adapter. The **plnfo** structure stores the RNID data to be set. The **plnfo** structure is defined in the **/usr/include/sys/hbaapi.h** file. The structure includes:

- wwn
- · unittype
- PortId
- NumberOfAttachedNodes
- IPVersion
- UDPort
- IPAddress
- reserved
- TopologyDiscoveryFlags

If successful, the SET RNID data in **plnfo** is sent to the adapter.

Parameters

handle Specifies the open file descriptor obtained from a successful call to the **open** subroutine.

plnfo Specifies the structure containing the information to be set or received from the **RNID** command

Return Values

Upon successful completion, the **HBA_SetRNIDMgmtInfo** subroutine returns a value of HBA_STATUS_OK, or a value of 0. If unsuccessful, a value of HBA_STATUS_ERROR, or a 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

HBA_STATUS_OK

A value of 0 on successful completion.

HBA_STATUS_ERROR

A value of 1 if an error occurred.

HBA_STATUS_ERROR_INVALID_HANDLEA value of 3 if there was an invalid file handle.

Related Information

"HBA_SendScsiInquiry Subroutine" on page 422, "HBA_SendReadCapacity Subroutine" on page 418, "HBA_SendCTPassThru Subroutine" on page 417, "HBA_SendReportLUNs Subroutine" on page 419, "HBA_SendRNID Subroutine" on page 421, and "HBA_GetRNIDMgmtInfo Subroutine" on page 414.

SCSI Adapter Device Driver in AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 2.

Special Files in AIX 5L Version 5.2 Files Reference.

SCSI Subsystem Overview, A Typical Initiator-Mode SCSI Driver Transaction Sequence, Required SCSI Adapter Device Driver ioctl Commands, Understanding the Execution of Initiator I/O Requests, SCSI Error Recovery, and Understanding the sc_buf Structure in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

hsearch, hcreate, or hdestroy Subroutine

Purpose

Manages hash tables.

Library

Standard C Library (libc.a)

Syntax

```
#include <search.h>
ENTRY *hsearch ( Item, Action)
ENTRY Item:
Action Action;
int hcreate ( NumberOfElements)
size t NumberOfElements;
void hdestroy ( )
```

Description

Attention: Do not use the hsearch, hcreate, or hdestroy subroutine in a multithreaded environment.

The **hsearch** subroutine searches a hash table. It returns a pointer into a hash table that indicates the location of the given item. The hsearch subroutine uses open addressing with a multiplicative hash function.

The hcreate subroutine allocates sufficient space for the table. You must call the hcreate subroutine before calling the **hsearch** subroutine. The *NumberOfElements* parameter is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

The **hdestroy** subroutine deletes the hash table. This action allows you to start a new hash table since only one table can be active at a time. After the call to the **hdestroy** subroutine, the data can no longer be considered accessible.

Parameters

Item

Identifies a structure of the type ENTRY as defined in the search.h file. It contains two pointers:

Item.key

Points to the comparison key. The key field is of the char type.

Item.data

Points to any other data associated with that key. The data field is of the void type.

Pointers to data types other than the **char** type should be declared to pointer-to-character.

Action Specifies the value of the Action enumeration parameter that indicates what is to be

done with an entry if it cannot be found in the table. Values are:

ENTER Enters the value of the *Item* parameter into the table at the appropriate point.

If the table is full, the **hsearch** subroutine returns a null pointer.

FIND D

Does not enter the value of the *Item* parameter into the table. If the value of the *Item* parameter cannot be found, the **hsearch** subroutine returns a null pointer. If the value of the *Item* parameter is found, the subroutine returns

the address of the item in the hash table.

NumberOfElements Provides an estimate of the maximum number of entries that the table contains.

Under some circumstances, the hcreate subroutine may actually make the table

larger than specified.

Return Values

The **hcreate** subroutine returns a value of 0 if it cannot allocate sufficient space for the table.

Related Information

The **bsearch** ("bsearch Subroutine" on page 102) subroutine, **Isearch** ("Isearch or Ifind Subroutine" on page 644) subroutine, **malloc** ("malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, or valloc Subroutine" on page 657) subroutine, **strcmp** subroutine, **tsearch** subroutine.

Searching and Sorting Example Program and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

hypot, hypotf, or hypotl Subroutine

Purpose

Computes the Euclidean distance function and complex absolute value.

Libraries

IEEE Math Library (**libm.a**)
System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>
double hypot ( x, y)
double x, y;
float hypotf (x, y)
float x;
float y;

long double hypotl (x, y)
long double x;
long double y;
```

Description

The **hypot**, **hypotf** and **hypotl** subroutines compute the value of the square root of $x^2 + y^2$ without undue overflow or underflow.

An application wishing to check for error situations should set the **errno** global variable to zero and call feclearexcept(FE ALL EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

- Specifies some double-precision floating-point value.
- Specifies some double-precision floating-point value.

Return Values

Upon successful completion, the hypot, hypotf and hypotl subroutines return the length of the hypotenuse of a right-angled triangle with sides of length x and y.

If the correct value would cause overflow, a range error occurs and the hypotf and hypotl subroutines return the value of the macro HUGE VALF and HUGE VALL, respectively.

If x or y is $\pm \ln f$, $+\ln f$ is returned (even if one of x or y is NaN).

If x or y is NaN, and the other is not $\pm \ln \ln n$, a NaN is returned.

If both arguments are subnormal and the correct result is subnormal, a range error may occur and the correct result is returned.

Error Codes

When using the libm.a (-lm) library, if the correct value overflows, the hypot subroutine returns a **HUGE VAL** value.

Note: (hypot (INF, value) and hypot (value, INF) are both equal to +INF for all values, even if value = NaN.

When using libmsaa.a (-Imsaa), if the correct value overflows, the hypot subroutine returns HUGE_VAL and sets the global variable errno to ERANGE.

These error-handling procedures may be changed with the **matherr** subroutine when using the **libmsaa.a** (-Imsaa) library.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, class, finite, isnan, or unordered Subroutines" on page 138.

The matherr ("matherr Subroutine" on page 665) subroutine, sqrt subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

math.h in AIX 5L Version 5.2 Files Reference.

iconv Subroutine

Purpose

Converts a string of characters in one character code set to another character code set.

Library

The iconv Library (libiconv.a)

Syntax

```
#include <iconv.h>
size_t iconv (CD, InBuf, InBytesLeft, OutBuf, OutBytesLeft)
iconv_t CD;
char **OutBuf, **InBuf;
size t *OutBytesLeft, *InBytesLeft;
```

Description

The iconv subroutine converts the string specified by the InBuf parameter into a different code set and returns the results in the *OutBuf* parameter. The required conversion method is identified by the *CD* parameter, which must be valid conversion descriptor returned by a previous, successful call to the iconv open subroutine.

On calling, the *InBytesLeft* parameter indicates the number of bytes in the *InBuf* buffer to be converted, and the OutBytesLeft parameter indicates the number of available bytes in the OutBuf buffer. These values are updated upon return so they indicate the new state of their associated buffers.

For state-dependent encodings, calling the **iconv** subroutine with the *InBuf* buffer set to null will reset the conversion descriptor in the CD parameter to its initial state. Subsequent calls with the InBuf buffer, specifying other than a null pointer, may cause the internal state of the subroutine to be altered a necessary.

Parameters

CD Specifies the conversion descriptor that points to the correct code set converter. InBuf Points to a buffer that contains the number of bytes in the InBytesLeft parameter to be

converted.

InBytesLeft Points to an integer that contains the number of bytes in the *InBuf* parameter.

OutBuf Points to a buffer that contains the number of bytes in the OutBytesLeft parameter that has

been converted.

OutBytesLeft 1 8 1 Points to an integer that contains the number of bytes in the *OutBuf* parameter.

Return Values

Upon successful conversion of all the characters in the InBuf buffer and after placing the converted characters in the OutBuf buffer, the iconv subroutine returns 0, updates the InBvtesLeft and OutBvtesLeft parameters, and increments the *InBuf* and *OutBuf* pointers. Otherwise, it updates the varibles pointed to by the parameters to indicate the extent to the conversion, returns the number of bytes still left to be converted in the input buffer, and sets the errno global variable to indicate the error.

Error Codes

If the iconv subroutine is unsuccessful, it updates the variables to reflect the extent of the conversion before it stopped and sets the **errno** global variable to one of the following values:

EILSEQ

Indicates an unusable character. If an input character does not belong to the input code set, no conversion is attempted on the unusable on the character. In InBytesLeft parameters indicates the bytes left to be converted, including the first byte of the unusable character. InBuf parameter points to the first byte of the unusable character sequence.

The values of OutBuf and OutBytesLeft are updated according to the number of bytes that were previously converted.

E2BIG

Indicates an output buffer overflow. If the OutBuf buffer is too small to contain all the converted characters, the character that causes the overflow is not converted. The InBytesLeft parameter indicates the bytes left to be converted (including the character that caused the overflow). The InBuf parameter points to the first byte of the characters left to convert.

EINVAL

Indicates the input buffer was truncated. If the original value of InBytesLeft is exhausted in the middle of a character conversion or shift/lock block, the InBytesLeft parameter indicates the number of bytes undefined in the character being converted.

If an input character of shift sequence is truncated by the InBuf buffer, no conversion is attempted on the truncated data, and the InBytesLeft parameter indicates the bytes left to be converted. The InBuf parameter points to the first bytes if the truncated sequence. The OutBuf and OutBytesLeft values are updated according to the number of characters that were previously0 converted. Because some encoding may have ambiguous data, the EINVAL return value has a special meaning at the end of stream conversion. As such, if a user detects an EOF character on a stream that is being converted and the last return code from the iconv subroutine was EINVAL, the iconv subroutine should be called again, with the same InBytesLeft parameter and the same character string pointed to by the InBuf parameter as when the EINVAL return occurred. As a result, the converter will either convert the string as is or declare it an unusable sequence (EILSEQ).

Files

/usr/lib/nls/loc/iconv/*

Contains code set converter methods.

Related Information

The iconv command, genxlt command.

The iconv close ("iconv close Subroutine") subroutine, iconv open ("iconv open Subroutine" on page 430) subroutine.

iconv_close Subroutine

Purpose

Closes a specified code set converter.

Library

iconv Library (libiconv.a)

Syntax

#include <iconv.h>

int iconv close (CD) iconv_t CD;

Description

The iconv_close subroutine closes a specified code set converter and deallocates any resources used by the converter.

Parameters

CD

Specifies the conversion descriptor to be closed.

Return Values

When successful, the iconv_close subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the errno global variable to indicate the error.

Error Codes

The following error code is defined for the **iconv_close** subroutine:

The conversion descriptor is not valid.

Related Information

The iconv ("iconv Subroutine" on page 427) subroutine, iconv_open ("iconv_open Subroutine") subroutine.

The **genxlt** command, **iconv** command.

National Language Support Overview and Converters Overview for Programming in AIX 5L Version 5.2 National Language Support Guide and Reference

iconv open Subroutine

Purpose

Opens a character code set converter.

Library

iconv Library (libiconv.a)

Syntax

```
#include <iconv.h>
```

```
iconv t iconv open ( ToCode, FromCode)
const char *ToCode, *FromCode;
```

Description

The iconv_open subroutine initializes a code set converter. The code set converter is used by the iconv subroutine to convert characters from one code set to another. The iconv open subroutine finds the converter that performs the character code set conversion specified by the FromCode and ToCode parameters, initializes that converter, and returns a conversion descriptor of type iconv t to identify the code set converter.

The iconv_open subroutine first searches the LOCPATH environment variable for a converter, using the two user-provided code set names, based on the file name convention that follows:

```
FromCode: "IBM-850"
ToCode: "IS08859-1"
conversion file: "IBM-850 IS08859-1"
```

The conversion file name is formed by concatenating the ToCode code set name onto the FromCode code set name, with an _ (underscore) between them.

The LOCPATH environment variable contains a list of colon-separated directory names. The system default for the LOCPATH environment variable is:

```
LOCPATH=/usr/lib/nls/loc
```

See Locales in AIX 5L Version 5.2 National Language Support Guide and Reference for more information on the LOCPATH environment variable.

The iconv_open subroutine first attempts to find the specified converter in an iconv subdirectory under any of the directories specified by the LOCPATH environment variable, for example, /usr/lib/nls/loc/iconv. If the iconv open subroutine cannot find a converter in any of these directories, it looks for a conversion table in an iconvTable subdirectory under any of the directories specified by the LOCPATH environment variable, for example, /usr/lib/nls/loc/iconvTable.

If the iconv_open subroutine cannot find the specified converter in either of these locations, it returns (iconv t) -1 to the calling process and sets the errno global variable.

The iconvTable directories are expected to contain conversion tables that are the output of the genxIt command. The conversion tables are limited to single-byte stateless code sets. See the "List of PC, ISO, and EBCDIC Code Set Converters" in AIX 5L Version 5.2 National Language Support Guide and Reference for more information.

If the named converter is found, the **iconv open** subroutine will perform the **load** subroutine operation and initialize the converter. A converter descriptor (iconv t) is returned.

Note: When a process calls the **exec** subroutine or a **fork** subroutine, all of the opened converters are discarded.

The **iconv open** subroutine links the converter function using the **load** subroutine, which is similar to the **exec** subroutine and effectively performs a run-time linking of the converter program. Since the iconv_open subroutine is called as a library function, it must ensure that security is preserved for certain programs. Thus, when the iconv open subroutine is called from a set root ID program (a program with permission —-s—-x), it will ignore the LOCPATH environment variable and search for converters only in the /usr/lib/nls/loc/iconv directory.

Parameters

ToCode Specifies the destination code set. FromCode Specifies the originating code set.

Return Values

A conversion descriptor (iconv t) is returned if successful. Otherwise, the subroutine returns -1, and the **errno** global variable is set to indicate the error.

Error Codes

The conversion specified by the FromCode and ToCode parameters is not supported by the **EINVAL**

implementation.

EMFILE The number of file descriptors specified by the OPEN MAX configuration variable is currently

open in the calling process.

Too many files are currently open in the system. **ENFILE**

ENOMEM Insufficient storage space is available.

Files

/usr/lib/nls/loc/iconv Contains loadable method converters.

/usr/lib/nls/loc/iconvTable Contains conversion tables for single-byte stateless code sets.

Related Information

The "iconv Subroutine" on page 427, "iconv_close Subroutine" on page 429.

The **genxlt** command, **iconv** command.

Code Sets for National Language Support, List of PC, ISO, and EBCDIC Code Set Converters, National Language Support Overview, and Converters Overview for Programming in AIX 5L Version 5.2 National Language Support Guide and Reference.

ilogbf, ilogbl, or ilogb Subroutine

Purpose

Returns an unbiased exponent.

Syntax

```
#include <math.h>
int ilogbf (x)
float x;
int ilogbl (x)
long double x;
int ilogb (x)
double x;
```

Description

The **ilogbf**, **ilogbl**, and **ilogb** subroutines return the exponent part of the x parameter. The return value is the integral part of $\log_r |x|$ as a signed integral value, for nonzero x, where r is the radix of the machine's floating-point arithmetic (r=2).

An application wishing to check for error situations should set thre errno global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the **ilogbf**, **ilogbl**, and **ilogb** subroutines return the exponent part of x as a signed integer value. They are equivalent to calling the corresponding logb function and casting the returned value to type int.

If *x* is 0, a domain error occurs, and the value **FP_ILOGB0** is returned.

If x is $\pm \ln f$, a domain error occurs, and the value {INT_MAX} is returned.

If x is a NaN, a domain error occurs, and the value **FP_ILOGBNAN** is returned.

If the correct value is greater than {INT MAX}, {INT MAX} is returned and a domain error occurs.

If the correct value is less than {INT_MIN}, {INT_MIN} is returned and a domain error occurs.

Related Information

"feclearexcept Subroutine" on page 223 and "fetestexcept Subroutine" on page 231.

math.h in AIX 5L Version 5.2 Files Reference.

imaxabs Subroutine

Purpose

Returns absolute value.

Syntax

```
#include <inttypes.h>
intmax t imaxabs (j)
intmax_t j;
```

Description

The **imaxabs** subroutine computes the absolute value of an integer j. If the result cannot be represented, the behavior is undefined.

Parameters

Specifies the value to be computed.

Return Values

The imaxabs subroutine returns the absolute value.

Related Information

The "imaxdiv Subroutine."

inttypes.h File in AIX 5L Version 5.2 Files Reference.

imaxdiv Subroutine

Purpose

Returns quotient and remainder.

Syntax

```
#include <inttypes.h>
imaxdiv_t imaxdiv (numer, denom)
intmax t numer;
intmax t denom;
```

Description

The **imaxdiv** subroutine computes *numer / denom* and *numer % denom* in a single operation.

Parameters

numerdenomSpecifies the numerator value to be computed.Specifies the denominator value to be computed.

Return Values

The **imaxdiv** subroutine returns a structure of type **imaxdiv_t**, comprising both the quotient and the remainder. The structure contains (in either order) the members *quot* (the quotient) and *rem* (the remainder), each of which has type **intmax_t**.

If either part of the result cannot be represented, the behavior is undefined.

Related Information

The "imaxabs Subroutine" on page 433.

inttypes.h File in AIX 5L Version 5.2 Files Reference.

IMAIXMapping Subroutine

Purpose

Translates a pair of Key and State parameters to a string and returns a pointer to this string.

Library

Input Method Library (IiblM.a)

Syntax

```
caddr_t IMAIXMapping(IMMap, Key, State, NBytes)
IMMap IMMap;
KeySym Key;
uint State;
int * NBytes;
```

Description

The **IMAIXMapping** subroutine translates a pair of *Key* and *State* parameters to a string and returns a pointer to this string.

This function handles the diacritic character sequence and Alt-NumPad key sequence.

Parameters

IMMap Identifies the keymap.

KeySpecifies the key symbol to which the string is mapped.StateSpecifies the state to which the string is mapped.

NBytes Returns the length of the returning string.

Return Values

If the length set by the *NBytes* parameter has a positive value, the **IMAIXMapping** subroutine returns a pointer to the returning string.

Note: The returning string is not null-terminated.

IMAuxCreate Callback Subroutine

Purpose

Tells the application program to create an auxiliary area.

Syntax

```
int IMAuxCreate( IM, AuxiliaryID, UData)
IMObject IM;
caddr_t *AuxiliaryID;
caddr_t UData;
```

Description

The IMAuxCreate subroutine is invoked by the input method of the operating system to create an auxiliary area. The auxiliary area can contain several different forms of data and is not restricted by the interface.

Most input methods display one auxiliary area at a time, but callbacks must be capable of handling multiple auxiliary areas.

This subroutine is provided by applications that use input methods.

Parameters

Indicates the input method instance. AuxiliaryID Identifies the newly created auxiliary area.

UData Identifies an argument passed by the IMCreate subroutine.

Return Values

On successful return of the IMAuxCreate subroutine, a newly created auxiliary area is set to the AuxiliaryID value and the IMError global variable is returned. Otherwise, the IMNoError value is returned.

Related Information

The IMCreate ("IMCreate Subroutine" on page 439) subroutine.

Input Methods, National Language Support Overview, and Using Callbacksin AIX 5L Version 5.2 National Language Support Guide and Reference

IMAuxDestroy Callback Subroutine

Purpose

Tells the application to destroy the auxiliary area.

Syntax

```
int IMAuxDestroy( IM, AuxiliaryID, UData)
IMObject IM;
caddr_t AuxiliaryID;
caddr_t UData;
```

Description

The IMAuxDestroy subroutine is called by the input method of the operating system to tell the application to destroy an auxiliary area.

This subroutine is provided by applications that use input methods.

Parameters

IM Indicates the input method instance. AuxiliaryID Identifies the auxiliary area to be destroyed. **UData** An argument passed by the IMCreate subroutine.

Return Values

If an error occurs, the IMAuxDestroy subroutine returns the IMError global variable. Otherwise, the **IMNoError** value is returned.

Related Information

The IMCreate ("IMCreate Subroutine" on page 439) subroutine.

Input Methods, and National Language Support Overview, and Using Callbacks in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMAuxDraw Callback Subroutine

Purpose

Tells the application program to draw the auxiliary area.

Syntax

int IMAuxDraw(IM, AuxiliaryID, AuxiliaryInformation, UData) IMObject IM; caddr_t AuxiliaryID; IMAuxInfo * AuxiliaryInformation; caddr_t UData;

Description

The IMAuxDraw subroutine is invoked by the input method to draw an auxiliary area. The auxiliary area should have been previously created.

This subroutine is provided by applications that use input methods.

Parameters

IM Indicates the input method instance. Identifies the auxiliary area. **AuxiliaryID** AuxiliaryInformation Points to the IMAuxInfo structure.

UData An argument passed by the IMCreate subroutine.

Return Values

If an error occurs, the IMAuxDraw subroutine returns the IMError global variable. Otherwise, the IMNoError value is returned.

Related Information

The IMAuxCreate ("IMAuxCreate Callback Subroutine" on page 435) subroutine, IMCreate ("IMCreate Subroutine" on page 439) subroutine.

Input Methods, National Language Support Overview, and Using Callbacks in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMAuxHide Callback Subroutine

Purpose

Tells the application program to hide an auxiliary area.

Syntax

```
int IMAuxHide( IM, AuxiliaryID, UData)
IMObject IM:
caddr t AuxiliaryID;
caddr_t UData;
```

Description

The IMAuxHide subroutine is called by the input method to hide an auxiliary area.

This subroutine is provided by applications that use input methods.

Parameters

IM Indicates the input method instance. AuxiliaryID Identifies the auxiliary area to be hidden.

UData An argument passed by the IMCreate subroutine.

Return Values

If an error occurs, the IMAuxHide subroutine returns the IMError global variable. Otherwise, the IMNoError value is returned.

Related Information

The IMAuxCreate ("IMAuxCreate Callback Subroutine" on page 435) subroutine, IMCreate ("IMCreate Subroutine" on page 439) subroutine.

Input Methods, National Language Support Overview, and Using Callbacks in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMBeep Callback Subroutine

Purpose

Tells the application program to emit a beep sound.

Syntax

```
int IMBeep( IM, Percent, UData)
IMObject IM;
int Percent;
caddr t UData;
```

Description

The **IMBeep** subroutine tells the application program to emit a beep sound.

This subroutine is provided by applications that use input methods.

Parameters

IM Indicates the input method instance.

Percent Specifies the beep level. The value range is from -100 to 100, inclusively. A -100 value means no beep.

UData An argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMBeep** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

Related Information

The **IMCreate** ("IMCreate Subroutine" on page 439) subroutine.

Input Methods, National Language Support Overview, and Using Callbacks in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMClose Subroutine

Purpose

Closes the input method.

Library

Input Method Library (liblM.a)

Syntax

void IMClose(IMfep)
IMFep IMfep;

Description

The **IMClose** subroutine closes the input method. Before the **IMClose** subroutine is called, all previously created input method instances must be destroyed with the **IMDestroy** subroutine, or memory will not be cleared.

Parameters

IMfep Specifies the input method.

Related Information

The **IMDestroy** ("IMDestroy Subroutine" on page 439) subroutine.

Input Method Overview and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMCreate Subroutine

Purpose

Creates one instance of an **IMObject** object for a particular input method.

Library

Input Method Library (liblM.a)

Syntax

IMObject IMCreate(IMfep, IMCallback, UData) IMFep IMfep; IMCallback *IMCallback; caddr t UData;

Description

The IMCreate subroutine creates one instance of a particular input method. Several input method instances can be created under one input method.

Parameters

IMfep Specifies the input method.

IMCallback Specifies a pointer to the caller-supplied **IMCallback** structure.

UData Optionally specifies an application's own information to the callback functions. With this

information, the application can avoid external references from the callback functions. The input method does not change this parameter, but merely passes it to the callback functions. The UData parameter is usually a pointer to the application data structure, which contains the

information about location, font ID, and so forth.

Return Values

The IMCreate subroutine returns a pointer to the created input method instance of type IMObject. If the subroutine is unsuccessful, a null value is returned and the imerrno global variable is set to indicate the error.

Related Information

The IMDestroy ("IMDestroy Subroutine") subroutine, IMFilter ("IMFilter Subroutine" on page 440) subroutine, IMLookupString ("IMLookupString Subroutine" on page 447) subroutine, IMProcess ("IMProcess Subroutine" on page 448) subroutine.

Input Methods and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMDestroy Subroutine

Purpose

Destroys an input method instance.

Library

Input Method Library (liblM.a)

Syntax

```
void IMDestroy( IM)
IMObject IM;
```

Description

The **IMDestroy** subroutine destroys an input method instance.

Parameters

IM

Specifies the input method instance to be destroyed.

Related Information

The **IMClose** ("IMClose Subroutine" on page 438) subroutine, **IMCreate** ("IMCreate Subroutine" on page 439) subroutine.

Input Methods and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMFilter Subroutine

Purpose

Determines if a keyboard event is used by the input method for internal processing.

Library

Input Method Library (liblM.a)

Syntax

```
int IMFilter(Im, Key, State, String, Length)
IMObect Im;
Keysym Key;
uint State, * Length;
caddr_t * String;
```

Description

The **IMFilter** subroutine is used to process a keyboard event and determine if the input method for this operating system uses this event. The return value indicates:

- The event is filtered (used by the input method) if the return value is **IMInputUsed**. Otherwise, the input method did not accept the event.
- Independent of the return value, a string may be generated by the keyboard event if pre-editing is complete.

Note: The buffer returned from the **IMFilter** subroutine is owned by the input method editor and can not continue between calls.

Parameters

Im Specifies the input method instance.Key Specifies the keysym for the event.

State Defines the state of the keysym. A value of 0 means that the keysym is not redefined.

String Holds the returned string if one exists. A null value means that no composed string is ready.

Length Defines the length of the input string. If the string is not null, returns the length.

Return Values

IMInputUsed The input method for this operating system filtered the event. **IMInputNotUsed** The input method for this operating system did not use the event.

Related Information

Input Methods and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMFreeKeymap Subroutine

Purpose

Frees resources allocated by the IMInitializeKeymap subroutine.

Library

Input Method Library (liblM.a)

Syntax

void IMFreeKeymap(IMMap) IMMap IMMap;

Description

The IMFreeKeymap subroutine frees resources allocated by the IMInitializeKeymap subroutine.

Parameters

ІММар Identifies the keymap.

Related Information

The IMInitializeKeymap ("IMInitializeKeymap Subroutine" on page 444) subroutine.

Input Methods and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMIndicatorDraw Callback Subroutine

Purpose

Tells the application program to draw the indicator.

Syntax

int IMIndicatorDraw(IM, IndicatorInformation, UData) IMObject IM; IMIndicatorInfo *IndicatorInformation; caddr_t UData;

Description

The IMIndicatorDraw callback subroutine is called by the input method when the value of the indicator is changed. The application program then draws the indicator.

This subroutine is provided by applications that use input methods.

Parameters

IM Indicates the input method instance.

IndicatorInformation Points to the IMIndicatorInfo structure that holds the current value of the

> indicator. The interpretation of this value varies among phonic languages. However, the input method provides a function to interpret this value.

UData An argument passed by the IMCreate subroutine.

Return Values

If an error happens, the **IMIndicatorDraw** subroutine returns the **IMError** global variable. Otherwise, the IMNoError value is returned.

Related Information

The IMCreate ("IMCreate Subroutine" on page 439) subroutine, IMIndicatorHide ("IMIndicatorHide Callback Subroutine") subroutine.

Input Methods, National Language Support Overview and Using Callbacks in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMIndicatorHide Callback Subroutine

Purpose

Tells the application program to hide the indicator.

Syntax

int IMIndicatorHide(IM, UData) IMObject IM; caddr_t UData;

Description

The **IMIndicatorHide** subroutine is called by the input method to tell the application program to hide the indicator.

This subroutine is provided by applications that use input methods.

Parameters

IM Indicates the input method instance.

UData Specifies an argument passed by the IMCreate subroutine.

Return Values

If an error occurs, the **IMIndicatorHide** subroutine returns the **IMError** global variable. Otherwise, the IMNoError value is returned.

Related Information

The IMCreate ("IMCreate Subroutine" on page 439) subroutine, IMIndicatorDraw ("IMIndicatorDraw Callback Subroutine" on page 441) subroutine.

Input Methods, National Language Support Overview and Understanding Callbacks in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMInitialize Subroutine

Purpose

Initializes the input method for a particular language.

Library

Input Method Library (liblM.a)

Syntax

IMFep IMInitialize(Name) char *Name;

Description

The IMInitialize subroutine initializes an input method. The IMCreate, IMFilter, and IMLookupString subroutines use the input method to perform input processing of keyboard events in the form of keysym state modifiers. The IMInitialize subroutine finds the input method that performs the input processing specified by the *Name* parameter and returns an Input Method Front End Processor (**IMFep**) descriptor.

Before calling any of the key event-handling functions, the application must create an instance of an IMObject object using the IMFep descriptor. Each input method can produce one or more instances of *IMObject* object with the **IMCreate** subroutine.

When the IMInitialize subroutine is called, strings returned from the input method are encoded in the code set of the locale. Each IMFep description inherits the code set of the locale when the input method is initialized. The locale setting does not change the code set of the IMFep description after it is created.

The IMInitialize subroutine calls the load subroutine to load a file whose name is in the form Name.im. The Name parameter is passed to the IMInitialize subroutine. The loadable input method file is accessed in the directories specified by the LOCPATH environment variable. The default location for loadable input-method files is the /usr/lib/nls/loc directory. If none of the LOCPATH directories contain the input method specified by the Name parameter, the default location is searched.

Note: All setuid and setgid programs will ignore the LOCPATH environment variable.

The name of the input method file usually corresponds to the locale name, which is in the form Language territory.codesest@modifier. In the environment, the modifier is in the form @im=modifier. The IMInitialize subroutine converts the @im= substring to @ when searching for loadable input-method files.

Parameters

Name

Specifies the language to be used. Each input method is dynamically linked to the application program.

Return Values

If IMInitialize succeeds, it returns an IMFep handle. Otherwise, null is returned and the imerrno global variable is set to indicate the error.

Files

/usr/lib/nls/loc

Contains loadable input-method files.

Related Information

The IMCreate ("IMCreate Subroutine" on page 439) subroutine.

Input Methods and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMInitializeKeymap Subroutine

Purpose

Initializes the keymap associated with a specified language.

Library

Input Method Library (liblM.a)

Syntax

IMMap IMInitializeKeymap(Name) char *Name;

Description

The IMInitializeKeymap subroutine initializes an input method keymap (imkeymap). The IMAIXMapping and **IMSimpleMapping** subroutines use the imkeymap to perform mapping of keysym state modifiers to strings. The IMInitializeKeymap subroutine finds the imkeymap that performs the keysym mapping and returns an imkeymap descriptor, IMMap. The strings returned by the imkeymap mapping functions are treated as unsigned bytes.

The applications that use input methods usually do not need to manage imkeymaps separately. The imkeymaps are managed internally by input methods.

The IMInitializeKeymap subroutine searches for an imkeymap file whose name is in the form Name.im. The Name parameter is passed to the IMInitializeKeymap subroutine. The imkeymap file is accessed in the directories specified by the LOCPATH environment variable. The default location for input method files is the /usr/lib/nls/loc directory. If none of the LOCPATH directories contain the keymap method specified by the Name parameter, the default location is searched.

Note: All setuid and setgid programs will ignore the LOCPATH environment variable.

The name of the imkeymap file usually corresponds to the locale name, which is in the form Language territory.codesest@modifier. In the AlXwindows environment, the modifier is in the form @im=modifier. The IMInitializeKeymap subroutine converts the @im= substring to @ (at sign) when searching for loadable input method files.

Parameters

Name

Specifies the name of the imkeymap.

Return Values

The IMInitializeKeymap subroutine returns a descriptor of type IMMap. Returning a null value indicates the occurrence of an error. The **IMMap** descriptor is defined in the **im.h** file as the **caddr t** structure. This descriptor is used for keymap manipulation functions.

Files

/usr/lib/nls/loc

Contains loadable input-method files.

Related Information

The IMFreeKeymap ("IMFreeKeymap Subroutine" on page 441), IMQueryLanguage ("IMQueryLanguage Subroutine" on page 450) subroutine.

Input Methods and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMIoctl Subroutine

Purpose

Performs a variety of control or query operations on the input method.

Library

Input Method Library (liblM.a)

Syntax

```
int IMIoctl( IM, Operation, Argument)
IMObject IM;
int Operation;
char *Argument;
```

Description

The IMloctI subroutine performs a variety of control or query operations on the input method specified by the IM parameter. In addition, this subroutine can be used to control the unique function of each language input method because it provides input method-specific extensions. Each input method defines its own function.

Parameters

IM Specifies the input method instance.

Operation

Specifies the operation.

Argument

The use of this parameter depends on which of the following operations is performed.

IM Refresh

Refreshes the text area, auxiliary areas, and indicator by calling the needed callback functions if these areas are not empty. The Argument parameter is not used.

IM_GetString

Gets the current pre-editing string. The Argument parameter specifies the address of the IMSTR structure supplied by the caller. The callback function is invoked to clear the pre-editing if it exists.

IM_Clear

Clears the text and auxiliary areas if they exist. If the Argument parameter is not a null value, this operation invokes the callback functions to clear the screen. The keyboard state remains the same.

IM Reset

Clears the auxiliary area if it currently exists. If the Argument parameter is a null value, this operation clears only the internal buffer of the input method. Otherwise, the **IMAuxHide** subroutine is called, and the input method returns to its initial state.

IM ChangeLength

Changes the maximum length of the pre-editing string.

IM_ChangeMode

Sets the Processing Mode of the input method to the mode specified by the Argument parameter. The valid value for *Argument* is:

IMNormalMode

Specifies the normal mode of pre-editing.

IMSuppressedMode

Suppresses pre-editing.

IM_QueryState

Returns the status of the text area, the auxiliary area, and the indicator. It also returns the beep status and the processing mode. The results are stored into the caller-supplied **IMQueryState** structure pointed to by the *Argument* parameter.

IM QueryText

Returns detailed information about the text area. The results are stored in the caller-supplied **IMQueryText** structure pointed to by the *Argument* parameter.

IM QueryAuxiliary

Returns detailed information about the auxiliary area. The results are stored in the caller-supplied **IMQueryAuxiliary** structure pointed to by the *Argument* parameter.

IM QueryIndicator

Returns detailed information about the indicator. The results are stored in the caller-supplied **IMQueryIndicator** structure pointed to by the *Argument* parameter.

IM QuervIndicatorString

Returns an indicator string corresponding to the current indicator. Results are stored in the caller-supplied **IMQueryIndicatorString** structure pointed to by the *Argument* parameter. The caller can request either a short or long form with the format member of the IMQueryIndicatorString structure.

IM SupportSelection

Informs the input method whether or not an application supports an auxiliary area selection list. The application must support selections inside the auxiliary area and determine how selections are displayed. If this operation is not performed, the input method assumes the application does not support an auxiliary area selection list.

Return Values

The IMIoctI subroutine returns a value to the IMError global variable that indicates the type of error encountered. Some error types are provided in the /usr/include/imerrno.h file.

Related Information

The IMFilter ("IMFilter Subroutine" on page 440) subroutine, IMLookupString ("IMLookupString Subroutine") subroutine, IMProcess ("IMProcess Subroutine" on page 448) subroutine.

Input Methods and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMLookupString Subroutine

Purpose

Maps a Key/State (key symbol/state) pair to a string.

Library

Input Method Library (liblM.a)

Syntax

```
int IMLookupString(Im, Key, State, String, Length)
IMObject Im;
KeySym Key;
uint State, * Length;
caddr t * String;
```

Description

The **IMLookupString** subroutine is used to map a Key/State pair to a localized string. It uses an internal input method keymap (imkeymap) file to map a keysym/modifier to a string. The string returned is encoded in the same code set as the locale of IMObject and IM Front End Processor.

Note: The buffer returned from the IMLookupString subroutine is owned by the input method editor and can not continue between calls.

Parameters

Im Specifies the input method instance. Kev Specifies the key symbol for the event.

State Defines the state for the event. A value of 0 means that the key is not redefined.

String Holds the returned string, if one exists. A null value means that no composed string is ready. Defines the length string on input. If the string is not null, identifies the length returned. Length

Return Values

IMError Error encountered.

IMReturnNothing No string or keysym was returned.

IMReturnString String returned.

Related Information

Input Methods and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMProcess Subroutine

Purpose

Processes keyboard events and language-specific input.

Library

Input Method Library (libIM.a)

Note: This subroutine will be removed in future releases. Use the **IMFilter** ("IMFilter Subroutine" on page 440) and **IMLookupString** ("IMLookupString Subroutine" on page 447) subroutines to process keyboard events.

Syntax

```
int IMProcess (IM, KeySymbol, State, String, Length)
IMObject IM;
KeySym KeySymbol;
uint State;
caddr_t * String;
uint * Length;
```

Description

This subroutine is a main entry point to the input method of the operating system. The **IMProcess** subroutine processes one keyboard event at a time. Processing proceeds as follows:

- Validates the *IM* parameter.
- Performs keyboard translation for all supported modifier states.
- Invokes internal function to do language-dependent processing.
- · Performs any necessary callback functions depending on the internal state.
- Returns to application, setting the String and Length parameters appropriately.

Parameters

IM Specifies the input method instance.

KeySymbol Defines the set of keyboard symbols that will be handled.

State Specifies the state of the keyboard.

String Holds the returned string. Returning a null value means that the input is used or discarded by the

input method.

Note: The String parameter is not a null-terminated string.

Length Stores the length, in bytes, of the *String* parameter.

Return Values

This subroutine returns the **IMError** global variable if an error occurs. The **IMerroo** global variable is set to indicate the error. Some of the variable values include:

IMError Error occurred during this subroutine.

IMTextAndAuxiliaryOff No text string in the Text area, and the Auxiliary area is not shown.

IMTextOn Text string in the Text area, but no Auxiliary area.

No text string in the Text area, and the Auxiliary area is shown. Text string in the Text area, and the Auxiliary is shown.

Related Information

The **IMClose** ("IMClose Subroutine" on page 438) subroutine, **IMCreate** ("IMCreate Subroutine" on page 439) subroutine **IMFilter** ("IMFilter Subroutine" on page 440) subroutine, **IMLookupString** ("IMLookupString Subroutine" on page 447) subroutine.

Input Methods and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMProcessAuxiliary Subroutine

Purpose

Notifies the input method of input for an auxiliary area.

Library

Input Method Library (liblM.a)

Syntax

Description

The IMProcessAuxiliary subroutine notifies the input method instance of input for an auxiliary area.

Parameters

IM Specifies the input method instance.AuxiliaryID Identifies the auxiliary area.

Button Specifies one of the following types of input:

IM_ABORT

Abort button is pushed.

IM_CANCEL

Cancel button is pushed.

IM ENTER

Enter button is pushed.

IM_HELP

Help button is pushed.

IM IGNORE

Ignore button is pushed.

IM_NO No button is pushed.

IM_OK OK button is pushed.

IM_RETRY

Retry button is pushed.

IM_SELECTED

Selection has been made. Only in this case do the *PanelRow, PanelColumn*,

ItemRow, and ItemColumn parameters have meaningful values.

IM_YES

Yes button is pushed.

PanelRow Indicates the panel on which the selection event occurred.

PanelColumn Indicates the panel on which the selection event occurred.

ItemRowIndicates the selected item.ItemColumnIndicates the selected item.

String Holds the returned string. If a null value is returned, the input is used or discarded by the input

method. Note that the String parameter is not a null-terminated string.

Length Stores the length, in bytes, of the String parameter.

Related Information

The IMAuxCreate ("IMAuxCreate Callback Subroutine" on page 435) subroutine.

Input Methods and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMQueryLanguage Subroutine

Purpose

Checks to see if the specified input method is supported.

Library

Input Method Library (IiblM.a)

Syntax

uint IMQueryLanguage(Name)
IMLanguage Name;

The IMQueryLanguage subroutine checks to see if the input method specified by the Name parameter is supported.

Parameters

Name Specifies the input method.

Return Values

The **IMQueryLanguage** subroutine returns a true value if the specified input method is supported, a false value if not.

Related Information

The IMClose ("IMClose Subroutine" on page 438) subroutine, IMInitialize ("IMInitialize Subroutine" on page 443) subroutine.

Input Methods, National Language Support Overview, Understanding Keyboard Mapping contains a list of supported languages in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMSimpleMapping Subroutine

Purpose

Translates a pair of KeySymbol and State parameters to a string and returns a pointer to this string.

Library

Input Method Library (liblM.a)

Syntax

```
caddr_t IMSimpleMapping (IMMap, KeySymbol, State, NBytes)
IMMap ;
KeySym KeySymbol;
uint State;
int * NBytes;
```

Description

Like the IMAIXMapping subroutine, the IMSimpleMapping subroutine translates a pair of KeySymbol and State parameters to a string and returns a pointer to this string. The parameters have the same meaning as those in the **IMAIXMapping** subroutine.

The IMSimpleMapping subroutine differs from the IMAIXMapping subroutine in that it does not support the diacritic character sequence or the Alt-NumPad key sequence.

Parameters

IMMap Identifies the keymap.

KeySymbol Key symbol to which the string is mapped. State Specifies the state to which the string is mapped.

NBytes Returns the length of the returning string.

Related Information

The IMAIXMapping ("IMAIXMapping Subroutine" on page 434) subroutine, IMFreeKeymap ("IMFreeKeymap Subroutine" on page 441) subroutine, IMInitializeKeymap ("IMInitializeKeymap Subroutine" on page 444) subroutine.

Input Method Overview and National Language Support Overview for Programming in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

IMTextCursor Callback Subroutine

Purpose

Asks the application to move the text cursor.

Syntax

```
int IMTextCursor(IM, Direction, Cursor, UData)
IMObject IM;
uint Direction;
int * Cursor:
caddr t UData;
```

Description

The IMTextCursor subroutine is called by the Input Method when the Cursor Up or Cursor Down key is input to the **IMFilter** and **IMLookupString** subroutines.

This subroutine sets the new display cursor position in the text area to the integer pointed to by the Cursor parameter. The cursor position is relative to the top of the text area. A value of -1 indicates the cursor should not be moved.

Because the input method does not know the actual length of the screen it always treats a text string as one-dimensional (a single line). However, in the terminal emulator, the text string sometimes wraps to the next line. The IMTextCursor subroutine performs this conversion from single-line to multiline text strings. When you move the cursor up or down, the subroutine interprets the cursor position on the text string relative to the input method.

This subroutine is provided by applications that use input methods.

Parameters

IM Indicates the Input Method instance.

Direction Specifies up or down.

Cursor Specifies the new cursor position or -1.

UData Specifies an argument passed by the IMCreate subroutine.

Return Values

If an error occurs, the IMTextCursor subroutine returns the IMError global variable. Otherwise, the IMNoError value is returned.

Related Information

The IMCreate ("IMCreate Subroutine" on page 439) subroutine, IMFilter ("IMFilter Subroutine" on page 440) subroutine, IMLookupString ("IMLookupString Subroutine" on page 447) subroutine, IMTextDraw ("IMTextDraw Callback Subroutine" on page 453) subroutine.

Input Methods, National Language Support Overview and Using Callbacks in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMTextDraw Callback Subroutine

Purpose

Tells the application program to draw the text string.

Syntax

int IMTextDraw(IM, TextInfo, UData) IMObject IM; IMTextInfo *TextInfo; caddr_t UData;

Description

The IMTextDraw subroutine is invoked by the Input Method whenever it needs to update the screen with its internal string. This subroutine tells the application program to draw the text string.

This subroutine is provided by applications that use input methods.

Parameters

Indicates the input method instance. TextInfo Points to the **IMTextInfo** structure.

UData An argument passed by the IMCreate subroutine.

Return Values

If an error occurs, the IMTextDraw subroutine returns the IMError global variable. Otherwise, the IMNoError value is returned.

Related Information

The IMCreate ("IMCreate Subroutine" on page 439) subroutine.

Input Methods, National Language Support Overview, and Using Callbacks in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMTextHide Callback Subroutine

Purpose

Tells the application program to hide the text area.

Syntax

int IMTextHide(IM, UData) IMObject IM; caddr_t UData;

Description

The IMTextHide subroutine is called by the input method when the text area should be cleared. This subroutine tells the application program to hide the text area.

This subroutine is provided by applications that use input methods.

Parameters

IM Indicates the input method instance.

UData Specifies an argument passed by the IMCreate subroutine.

Return Values

If an error occurs, the IMTextHide subroutine returns an IMError value. Otherwise, an IMNoError value is returned.

Related Information

The IMTextDraw ("IMTextDraw Callback Subroutine" on page 453) subroutine.

Input Methods, National Language Support Overview, and Using Callbacks in AIX 5L Version 5.2 National Language Support Guide and Reference.

IMTextStart Callback Subroutine

Purpose

Notifies the application program of the length of the pre-editing space.

Syntax

```
int IMTextStart( IM, Space, UData)
IMObject IM;
int *Space;
caddr_t UData;
```

Description

The IMTextStart subroutine is called by the input method when the pre-editing is started, but prior to calling the IMTextDraw callback subroutine. This subroutine notifies the input method of the length, in terms of bytes, of pre-editing space. It sets the length of the available space (>=0) on the display to the integer pointed to by the Space parameter. A value of -1 indicates that the pre-editing space is dynamic and has no limit.

This subroutine is provided by applications that use input methods.

Parameters

IM Indicates the input method instance. Space Maximum length of pre-editing string.

UData An argument passed by the IMCreate subroutine.

Related Information

The IMCreate ("IMCreate Subroutine" on page 439) subroutine, IMTextDraw ("IMTextDraw Callback Subroutine" on page 453) subroutine.

Input Methods, Using Callbacks, and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

inet_aton Subroutine

Purpose

Converts an ASCII string into an Internet address.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton ( CharString, InternetAddr)
char * CharString;
struct in_addr * InternetAddr;
```

Description

The **inet_aton** subroutine takes an ASCII string representing the Internet address in dot notation and converts it into an Internet address.

All applications containing the **inet_aton** subroutine must be compiled with **_BSD** set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Parameters

CharString Contains the ASCII string to be converted to an Internet address.

InternetAddr Contains the Internet address that was converted from the ASCII string.

Return Values

Upon successful completion, the **inet_aton** subroutine returns 1 if *CharString* is a valid ASCII representation of an Internet address.

The **inet_aton** subroutine returns 0 if *CharString* is not a valid ASCII representation of an Internet address.

Files

/etc/hostsContains host names./etc/networksContains network names.

Related Information

The endhostent subroutine, endnetent subroutine, gethostbyaddr subroutine, gethostbyname subroutine, getnetbyaddr subroutine, getnetbyname subroutine, getnetent subroutine, inet_addr subroutine, inet_lnaof subroutine, inet_makeaddr subroutine, inet_network subroutine, inet_ntoa subroutine, sethostent subroutine, setnetent subroutine.

Sockets Overview and Network Address Translation in *AIX 5L Version 5.2 Communications Programming Concepts.*

initgroups Subroutine

Purpose

Initializes supplementary group ID.

Library

Standard C Library (libc.a)

Syntax

int initgroups (User, BaseGID)
const char *User;
int BaseGID;

Description

Attention: The **initgroups** subroutine uses the **getgrent** and **getpwent** family of subroutines. If the program that invokes the **initgroups** subroutine uses any of these subroutines, calling the **initgroups** subroutine overwrites the static storage areas used by these subroutines.

The **initgroups** subroutine reads the defined group membership of the specified *User* parameter and sets the supplementary group ID of the current process to that value. The *BaseGID* parameter is always included in the supplementary group ID. The supplementary group is normally the principal user's group. If the user is in more than **NGROUPS_MAX** groups, set in the **limits.h** file, only **NGROUPS_MAX** groups are set, including the *BaseGID* group.

Parameters

User Identifies a user.

BaseGID Specifies an additional group to include in the group set.

Return Values

- Indicates that the subroutine was success.
- -1 Indicates that the subroutine failed. The **errno** global variable is set to indicate the error.

Related Information

The **getgid** ("getgid, getegid or gegidx Subroutine" on page 317) subroutine, **getgrent**, **getgrgid**, **getgrnam**, **putgrent**, **setgrent**, or **endgrent** ("getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine" on page 318) subroutine, **getgroups** ("getgroups Subroutine" on page 325) subroutine, **setgroups** subroutine.

The **groups** command, **setgroups** command.

List of Security and Auditing Subroutines, Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

initialize Subroutine

Purpose

Performs printer initialization.

Library

None (provided by the formatter).

Syntax

#include <piostruct.h>

int initialize ()

Description

The **initialize** subroutine is invoked by the formatter driver after the **setup** subroutine returns.

If the -j flag passed from the qprt command has a nonzero value (true), the initialize subroutine uses the piocmdout subroutine to send a command string to the printer. This action initializes the printer to the proper state for printing the file. Any variables referenced by the command string should be the attribute values from the database, overridden by values from the command line.

If the -j flag passed from the qprt command has a nonzero value (true), any necessary fonts should be downloaded.

Return Values

Indicates a successful operation.

If the initialize subroutine detects an error, it uses the piomsgout subroutine to invoke an error message. It then invokes the **pioexit** subroutine with a value of **PIOEXITBAD**.

Note: If either the piocmdout or piogetstr subroutine detects an error, it issues its own error messages and terminates the print job.

Related Information

The piocmdout subroutine, pioexit subroutine, piogetstr subroutine, piomsgout subroutine, setup subroutine.

Adding a New Printer Type to Your System, Printer Addition Management Subsystem: Programming Overview, Understanding Embedded References in Printer Attribute Strings in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

Example of Print Formatter in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

insque or remque Subroutine

Purpose

Inserts or removes an element in a queue.

Library

Standard C Library (libc.a)

Syntax

#include <search.h>

```
insque ( Element, Pred)
void *Element, *Pred;
remque (Element)
void *Element;
```

The **insque** and **remque** subroutines manipulate queues built from double-linked lists. Each element in the queue must be in the form of a **qelem** structure. The **next** and **prev** elements of that structure must point to the elements in the queue immediately before and after the element to be inserted or deleted.

The **insque** subroutine inserts the element pointed to by the *Element* parameter into a queue immediately after the element pointed to by the *Pred* parameter.

The **remque** subroutine removes the element defined by the *Element* parameter from a queue.

Parameters

Pred Points to the element in the queue immediately before the element to be inserted or deleted.

Element Points to the element in the queue immediately after the element to be inserted or deleted.

Related Information

Searching and Sorting Example Program in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

install_lwcf_handler Subroutine

Purpose

Registers the signal handler to dump a lightweight core file for signals that normally cause the generation of a core file.

Library

PTools Library (libptools_ptr.a)

Syntax

void install_lwcf_handler (void);

Description

The **install_lwcf_handler** subroutine registers the signal handler to dump a lightweight core file for signals that normally cause a core file to be generated. The format of lightweight core files complies with the Parallel Tools Consortium Lightweight Core File Format.

The **install_lwcf_handler** subroutine uses the **LIGHTWEIGHT_CORE** environment variable to determine the target lightweight core file. If the **LIGHTWEIGHT_CORE** environment variable is defined, a lightweight core file will be generated. Otherwise, a normal core file will be generated.

If the **LIGHTWEIGHT_CORE** environment variable is defined without a value, the lightweight core file is assigned the default file name **lw_core** and is created under the current working directory if it does not already exist.

If the LIGHTWEIGHT CORE environment variable is defined with a value of STDERR, the lightweight core file is output to the standard error output device of the process. Keyword STDERR is not case-sensitive.

If the LIGHTWEIGHT CORE environment variable is defined with the value of a character string other than STDERR, the string is used as a path name for the lightweight core file generated.

If the target lightweight core file already exists, the traceback information is appended to the file.

The install_lwcf_handler subroutine can be called directly from an application to register the signal handler. Alternatively, linker option -binitfini:install_lwcf_handler can be used when linking an application, which specifies to execute the install lwcf handler subroutine when the application is initialized. The advantage of the second method is that the application code does not need to change to invoke the install_lwcf_handler subroutine.

Related Information

The **mt** trce and sigaction subroutines.

ioctl, ioctlx, ioctl32, or ioctl32x Subroutine

Purpose

Performs control functions associated with open file descriptors.

Library

Standard C Library (libc.a)

BSD Library (libbsd.a)

Syntax

```
#include <sys/ioctl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stropts.h>
int ioctl (FileDescriptor, Command, Argument)
int FileDescriptor, Command;
void * Argument;
int ioctlx (FileDescriptor, Command, Argument, Ext)
int FileDescriptor, Command;
void *Argument;
int Ext;
int ioct132 (FileDescriptor, Command, Argument)
int FileDescriptor, Command;
unsigned int Argument;
int ioct132x (FileDescriptor, Command, Argument, Ext)
int FileDescriptor, Command;
unsigned int Argument;
unsigned int Ext;
```

The **ioctl** subroutine performs a variety of control operations on the object associated with the specified open file descriptor. This function is typically used with character or block special files, sockets, or generic device support such as the termio general terminal interface.

The control operation provided by this function call is specific to the object being addressed, as are the data type and contents of the Argument parameter. The ioctlx form of this function can be used to pass an additional extension parameter to objects supporting it. The ioct132 and ioct132x forms of this function behave in the same way as ioctl and ioctlx, but allow 64-bit applications to call the ioctl routine for an object that does not normally work with 64-bit applications.

Performing an ioctl function on a file descriptor associated with an ordinary file results in an error being returned.

Parameters

Specifies the open file descriptor for which the control operation is to be performed. FileDescriptor Command

Specifies the control function to be performed. The value of this parameter depends on

which object is specified by the FileDescriptor parameter.

Specifies additional information required by the function requested in the Command Argument

parameter. The data type of this parameter (a void pointer) is object-specific, and is typically used to point to an object device-specific data structure. However, in some

device-specific instances, this parameter is used as an integer.

Ext Specifies an extension parameter used with the ioctlx subroutine. This parameter is

passed on to the object associated with the specified open file descriptor. Although normally of type int, this parameter can be used as a pointer to a device-specific

structure for some devices.

File Input/Output (FIO) ioctl Command Values

A number of file input/output (FIO) ioctl commands are available to enable the ioctl subroutine to function similar to the fcntl subroutine:

FIOCLEX and FIONCLEX

Manipulate the close-on-exec flag to determine if a file descriptor should be closed as part of the normal processing of the **exec** subroutine. If the flag is set, the file descriptor is closed. If the flag is clear, the file descriptor is left open.

The following code sample illustrates the use of the fcntl subroutine to set and clear the close-on-exec flag:

```
/* set the close-on-exec flag for fd1 */
fcntl(fd1,F SETFD,FD CLOEXEC);
/* clear the close-on-exec flag for fd2 */
fcnt1(fd2,F SETFD,0);
```

Although the fcntl subroutine is normally used to set the close-on-exec flag, the ioctl subroutine may be used if the application program is linked with the Berkeley Compatibility Library (libbsd.a) or the Berkeley Thread Safe Library (libbsd_r.a) (4.2.1 and later versions). The following loctl code fragment is equivalent to the preceding fcntl fragment:

```
/* set the close-on-exec flag for fd1 */
ioctl(fd1,FIOCLEX,0);
/* clear the close-on-exec flag for fd2 */
ioctl(fd2,FIONCLEX,0);
```

The third parameter to the ioctl subroutine is not used for the FIOCLEX and FIONCLEX ioctl commands.

FIONBIO

Enables nonblocking I/O. The effect is similar to setting the **O_NONBLOCK** flag with the **fcntl** subroutine. The third parameter to the **ioctl** subroutine for this command is a pointer to an integer that indicates whether nonblocking I/O is being enabled or disabled. A value of 0 disables non-blocking I/O. Any nonzero value enables nonblocking I/O. A sample code fragment follows:

```
int flag;
/* enable NBIO for fd1 */
flag = 1;
ioctl(fd1,FIONBIO,&flag);
/* disable NBIO for fd2 */
flag = 0;
ioctl(fd2,FIONBIO,&flag);
```

Determines the number of bytes that are immediately available to be read on a file descriptor. The third parameter to the **ioctl** subroutine for this command is a pointer to an integer variable where the byte count is to be returned. The following sample code illustrates the proper use of the **FIONREAD** ioctl command:

```
int nbytes;
ioctl(fd,FIONREAD,&nbytes);
```

Enables a simple form of asynchronous I/O notification. This command causes the kernel to send **SIGIO** signal to a process or a process group when I/O is possible. Only sockets, ttys, and pseudo-ttys implement this functionality.

The third parameter of the **ioctl** subroutine for this command is a pointer to an integer variable that indicates whether the asynchronous I/O notification should be enabled or disabled. A value of 0 disables I/O notification; any nonzero value enables I/O notification. A sample code segment follows:

```
int flag;
/* enable ASYNC on fd1 */
flag = 1;
ioctl(fd, FIOASYNC,&flag);
/* disable ASYNC on fd2 */
flag = 0;
ioctl(fd,FIOASYNC,&flag);
```

Sets the recipient of the **SIGIO** signals when asynchronous I/O notification (**FIOASYNC**) is enabled. The third parameter to the **ioctl** subroutine for this command is a pointer to an integer that contains the recipient identifier. If the value of the integer pointed to by the third parameter is negative, the value is assumed to be a process group identifier. If the value is positive, it is assumed to be a process identifier.

Sockets support both process groups and individual process recipients, while ttys and psuedo-ttys support only process groups. Attempts to specify an individual process as the recipient will be converted to the process group to which the process belongs. The following code example illustrates how to set the recipient identifier:

```
int owner;
owner = -getpgrp();
ioctl(fd,FIOSETOWN,&owner);
```

Note: In this example, the asynchronous I/O signals are being enabled on a process group basis. Therefore, the value passed through the owner parameter must be a negative number.

The following code sample illustrates enabling asynchronous I/O signals to an individual process:

```
int owner;
owner = getpid();
ioctl(fd,FIOSETOWN,&owner);
```

FIONREAD

FIOASYNC

FIOSETOWN

FIOGETOWN Determines the current recipient of the asynchronous I/O signals of an object

> that has asynchronous I/O notification (FIOASYNC) enabled. The third parameter to the ioctl subroutine for this command is a pointer to an integer

used to return the owner ID. For example:

ioctl(fd,FIOGETOWN,&owner);

If the owner of the asynchronous I/O capability is a process group, the value returned in the reference parameter is negative. If the owner is an individual

process, the value is positive.

Return Values

If the **ioctl** subroutine fails, a value of -1 is returned. The **errno** global variable is set to indicate the error.

The **ioctl** subroutine fails if one or more of the following are true:

EBADF The FileDescriptor parameter is not a valid open file

descriptor.

EFAULT The Argument or Ext parameter is used to point to data

outside of the process address space.

EINTR A signal was caught during the **ioctl** or **ioctlx** subroutine

and the process had not enabled re-startable subroutines

for the signal.

EINTR A signal was caught during the ioctl , ioctlx , ioctl32 , or

ioct132x subroutine and the process had not enabled

re-startable subroutines for the signal.

EINVAL The Command or Argument parameter is not valid for the

specified object.

ENOTTY The FileDescriptor parameter is not associated with an

object that accepts control functions.

ENODEV The FileDescriptor parameter is associated with a valid

character or block special file, but the supporting device

driver does not support the ioctl function.

ENXIO The FileDescriptor parameter is associated with a valid

character or block special file, but the supporting device

driver is not in the configured state.

Object-specific error codes are defined in the documentation for associated objects.

Related Information

The **ddioctl** device driver entry point and the **fp_ioctl** kernel service in AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems.

The Special Files Overview in AIX 5L Version 5.2 Files Reference.

The Input and Output Handling Programmer's Overview, the tty Subsystem Overview, in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

The Sockets Overview and Understanding Socket Data Transfer in AIX 5L Version 5.2 Communications Programming Concepts.

isblank Subroutine

Purpose

Tests for a blank character.

Syntax

```
#include <ctype.h>
int isblank (c)
int c;
```

Description

The **isblank** subroutine tests whether the c parameter is a character of class **blank** in the program's current locale.

The c parameter is a type int, the value of which the application shall ensure is a character representable as an unsigned char or equal to the value of the macro EOF. If the parameter has any other value, the behavior is undefined.

Parameters

Specifies the character to be tested.

Return Values

The **isblank** subroutine returns nonzero if *c* is a <blank>; otherwise, it returns 0.

Related Information

The "ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines" on page 168.

setlocale Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

isendwin Subroutine

Purpose

Determines whether the endwin subroutine was called without any subsequent refresh calls.

Library

Curses Library (libcurses.a)

Syntax

#include <curses.h> isendwin()

Description

The isendwin subroutine determines whether the endwin subroutine was called without any subsequent refresh calls. If the endwin was called without any subsequent calls to the wrefresh or doupdate subroutines, the isendwin subroutine returns TRUE.

Return Values

TRUE Indicates the endwin subroutine was called without any subsequent calls to the wrefresh or doupdate

FALSE Indicates subsequest calls to the refresh subroutines.

Related Information

The doupdate subroutine, endwin subroutine, wrefresh subroutine.

Curses Overview for Programming, Initializing Curses, List of Curses Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

isfinite Macro

Purpose

Tests for finite value.

Syntax

```
#include <math.h>
int isfinite (x)
real-floating x;
```

Description

The **isfinite** macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). An argument represented in a format wider than its semantic type is converted to its semantic type. Determination is based on the type of the argument.

Parameters

x Specifies the value to be tested.

Return Values

The isfinite macro returns a nonzero value if its argument has a finite value.

Related Information

"fpclassify Macro" on page 266, "isinf Subroutine" on page 466, "class, _class, finite, isnan, or unordered Subroutines" on page 138, "isnormal Macro" on page 468.

The signbit Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

math.h in AIX 5L Version 5.2 Files Reference.

isgreater Macro

Purpose

Tests if x is greater than y.

Syntax

```
#include <math.h>
int isgreater (x, y)
real-floating x;
real-floating y;
```

The isgreater macro determines whether its first argument is greater than its second argument. The value of **isgreater**(x, y) is equal to (x) > (y); however, unlike (x) > (y), **isgreater**(x, y) does not raise the invalid floating-point exception when x and y are unordered.

Parameters

- Specifies the first value to be compared. V Specifies the first value to be compared.
- **Return Values**

Upon successful completion, the **isgreater** macro returns the value of (x) > (y).

If x or y is NaN, 0 is returned.

Related Information

"isgreaterequal Subroutine," "isless Macro" on page 466, "islessequal Macro" on page 467, "islessgreater Macro" on page 468, and "isunordered Macro" on page 469.

math.h in AIX 5L Version 5.2 Files Reference.

isgreaterequal Subroutine

Purpose

Tests if x is greater than or equal to y.

Syntax

```
#include <math.h>
int isgreaterequal (x, y)
real-floating x;
real-floating y;
```

Description

The isgreaterequal macro determines whether its first argument is greater than or equal to its second argument. The value of **isgreaterequal** (x, y) is equal to (x) >= (y); however, unlike (x) >= (y), **isgreateregual** (x, y) does not raise the invalid floating-point exception when x and y are unordered.

Parameters

```
Specifies the first value to be compared.
Specifies the second value to be compared.
```

Return Values

Upon successful completion, the **isgreaterequal** macro returns the value of (x) >= (y).

If x or y is NaN, 0 is returned.

Related Information

"isgreater Macro" on page 464, "isless Macro," "islessequal Macro" on page 467, "islessgreater Macro" on page 468, and "isunordered Macro" on page 469.

math.h in AIX 5L Version 5.2 Files Reference.

isinf Subroutine

Purpose

Tests for infinity.

Syntax

```
#include <math.h>
int isinf (x)
real-floating x;
```

Description

The **isinf** macro determines whether its argument value is an infinity (positive or negative). An argument represented in a format wider than its semantic type is converted to its semantic type. Determination is based on the type of the argument.

Parameters

x Specifies the value to be checked.

Return Values

The isinf macro returns a nonzero value if its argument has an infinite value.

Related Information

"fpclassify Macro" on page 266, "isfinite Macro" on page 464, "class, _class, finite, isnan, or unordered Subroutines" on page 138, "isnormal Macro" on page 468.

The signbit Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

math.h in AIX 5L Version 5.2 Files Reference.

isless Macro

Purpose

Tests if x is less than y.

Syntax

```
#include <math.h>
int isless (x, y)
real-floating x;
real-floating y;
```

The isless macro determines whether its first argument is less than its second argument. The value of **isless**(x, y) is equal to (x) < (y); however, unlike (x) < (y), **isless**(x, y) does not raise the invalid floating-point exception when x and y are unordered.

Parameters

- Specifies the first value to be compared.
- V Specifies the second value to be compared.

Return Values

Upon successful completion, the **isless** macro returns the value of (x) < (y).

If x or y is NaN, 0 is returned.

Related Information

"isgreater Macro" on page 464, "isgreateregual Subroutine" on page 465, "islessegual Macro," "islessgreater Macro" on page 468, and "isunordered Macro" on page 469.

math.h in AIX 5L Version 5.2 Files Reference.

islessequal Macro

Purpose

Tests if x is less than or equal to y.

Syntax

```
#include <math.h>
int islessequal (x, y)
real-floating x;
real-floating y;
```

Description

The **islessequal** macro determines whether its first argument is less than or equal to its second argument. The value of **islessequal**(x, y) is equal to $(x) \le (y)$; however, unlike $(x) \le (y)$, **islessequal**(x, y) does not raise the invalid floating-point exception when x and y are unordered.

Parameters

```
Specifies the first value to be compared.
Specifies the second value to be compared.
```

Return Values

Upon successful completion, the **islessequal** macro returns the value of $(x) \le (y)$.

If x or y is NaN, 0 is returned.

Related Information

"isgreater Macro" on page 464, "isgreaterequal Subroutine" on page 465, "islessequal Macro" on page 467, "islessgreater Macro," and "isunordered Macro" on page 469.

math.h in AIX 5L Version 5.2 Files Reference.

islessgreater Macro

Purpose

Tests if x is less than or greater than y.

Syntax

```
#include <math.h>
int islessgreater (x, y)
real-floating x;
real-floating y;
```

Description

The **islessgreater** macro determines whether its first argument is less than or greater than its second argument. The **islessgreater**(x, y) macro is similar to (x) < (y) || (x) > (y); however, **islessgreater**(x, y) does not raise the invalid floating-point exception when x and y are unordered (nor does it evaluate x and y twice).

Parameters

- x Specifies the first value to be compared.
- y Specifies the second value to be compared.

Return Values

Upon successful completion, the **islessgreater** macro returns the value of $(x) < (y) \mid |(x) > (y)$.

If x or y is NaN, 0 is returned.

Related Information

"isgreater Macro" on page 464, "isgreaterequal Subroutine" on page 465, "isless Macro" on page 466, "islessequal Macro" on page 467, and "isunordered Macro" on page 469.

math.h in AIX 5L Version 5.2 Files Reference.

isnormal Macro

Purpose

Tests for a normal value.

Syntax

```
#include <math.h>
int isnormal (x)
real-floating x;
```

The isnormal macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN) or not. An argument represented in a format wider than its semantic type is converted to its semantic type. Determination is based on the type of the argument.

Parameters

Specifies the value to be tested.

Return Values

The **isnormal** macro returns a nonzero value if its argument has a normal value.

Related Information

"fpclassify Macro" on page 266, "isfinite Macro" on page 464, "isinf Subroutine" on page 466, "class, class, finite, isnan, or unordered Subroutines" on page 138.

The signbit Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

math.h in AIX 5L Version 5.2 Files Reference.

isunordered Macro

Purpose

Tests if arguments are unordered.

Syntax

```
#include <math.h>
int isunordered (x, y)
real-floating x;
real-floating y;
```

Description

The isunordered macro determines whether its arguments are unordered.

Parameters

- Specifies the first value in the order.
- Specifies the second value in the order. У

Return Values

Upon successful completion, the isunordered macro returns 1 if its arguments are unordered, and 0 otherwise.

If x or y is NaN, 0 is returned.

Related Information

"isgreater Macro" on page 464, "isgreaterequal Subroutine" on page 465, "isless Macro" on page 466, "islessequal Macro" on page 467, and "islessgreater Macro" on page 468.

iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, or iswxdigit Subroutine

Purpose

Tests a wide character for membership in a specific character class.

Library

Standard C Library (libc.a)

Syntax

#include <wchar.h> int iswalnum (WC) wint_t WC; int iswalpha (WC) wint_t WC; int iswcntrl (WC) wint t WC; int iswdigit (WC) wint t WC; int iswgraph (WC) wint t WC; int iswlower (WC) wint t WC; int iswprint (WC) wint_t WC; int iswpunct (WC) wint t WC; int iswspace (WC) wint_t WC; int iswupper (WC) wint_t WC; int iswxdigit (WC) wint_t WC;

Description

The **isw** subroutines check the character class status of the wide character code specified by the *WC* parameter. Each subroutine tests to see if a wide character is part of a different character class. If the wide character is part of the character class, the **isw** subroutine returns true; otherwise, it returns false.

Each subroutine is named by adding the **isw** prefix to the name of the character class that the subroutine tests. For example, the **iswalpha** subroutine tests whether the wide character specified by the *WC* parameter is an alphabetic character. The character classes are defined as follows:

alnum Alphanumeric character. **alpha** Alphabetic character.

cntrl Control character. No characters in the **alpha** or **print** classes are included.

digit Numeric digit character.

graph Graphic character for printing, not including the space character or cntrl characters. Includes all

characters in the digit and punct classes.

lower Lowercase character. No characters in **cntrl**, **digit**, **punct**, or **space** are included.

print Print character. All characters in the graph class are included, but no characters in cntrl are included.

punct Punctuation character. No characters in the alpha, digit, or cntrl classes, or the space character are

included.

space Space characters. Uppercase character. upper xdigit Hexadecimal character.

Parameters

WC Specifies a wide character for testing.

Return Values

If the wide character tested is part of the particular character class, the isw subroutine returns a nonzero value; otherwise it returns a value of 0.

Related Information

The **iswctype** subroutine, ("iswctype or is_wctype Subroutine" on page 472)**setlocale** subroutine, towlower subroutine, towupper subroutine wctype subroutine.

Subroutines, Example Programs, and Libraries, Wide Character Classification Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

iswblank Subroutine

Purpose

Tests for a blank wide-character code.

Syntax

#include <wctype.h> int iswblank (wc) wint_t wc;

Description

The **iswblank** subroutine tests whether the wc parameter is a wide-character code representing a character of class blank in the program's current locale.

The wc parameter is a wint t, the value of which the application ensures is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the parameter has any other value, the behavior is undefined.

Parameters

WC Specifies the value to be tested.

Return Values

The **iswblank** subroutine returns a nonzero value if the wc parameter is a blank wide-character code; otherwise, it returns a 0.

Related Information

"iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, or iswxdigit Subroutine" on page 470 and "iswctype or is_wctype Subroutine."

setlocale Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

wctype.h in AIX 5L Version 5.2 Files Reference.

iswctype or is wctype Subroutine

Purpose

Determines properties of a wide character.

Library

Standard C Library (libc. a)

Syntax

```
#include <wchar.h>
int iswctype ( WC, Property)
wint t WC;
wctype t Property;
int is_wctype ( WC, Property)
wint t WC;
wctype_t Property;
```

Description

The **iswctype** subroutine tests the wide character specified by the WC parameter to determine if it has the property specified by the *Property* parameter. The **iswctype** subroutine is defined for the wide-character null value and for values in the character range of the current code set, defined in the current locale. The is wctype subroutine is identical to the iswctype subroutine.

The **iswctype** subroutine adheres to X/Open Portability Guide Issue 5.

Parameters

WC Specifies the wide character to be tested. **Property** Specifies the property for which to test.

Return Values

If the WC parameter has the property specified by the Property parameter, the iswctype subroutine returns a nonzero value. If the value specified by the WC parameter does not have the property specified by the *Property* parameter, the **iswctype** subroutine returns a value of zero. If the value specified by the WC parameter is not in the subroutine's domain, the result is undefined. If the value specified by the Property parameter is not valid (that is, not obtained by a call to the wctype subroutine, or the Property parameter has been invalidated by a subsequent call to the setlocale subroutine that has affected the LC_CTYPE category), the result is undefined.

Related Information

The "iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, or iswxdigit Subroutine" on page 470.

Subroutines, Example Programs, and Libraries, Wide Character Classification Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

icode Subroutines

Purpose

Perform string conversion on 8-bit processing codes.

Library

Standard C Library (libc.a)

Syntax

```
#include <jcode.h>
char *jistosj( String1, String2)
char *String1, *String2;
char *jistouj(String1, String2)
char *String1, *String2;
char *sjtojis(String1, String2)
char *String1, *String2;
char *sjtouj(String1, String2)
char *String1, *String2;
char *ujtojis(String1, String2)
char *String1, *String2;
char *ujtosj(String1, String2)
char *String1, *String2;
char *cjistosj(String1, String2)
char *String1, *String2;
char *cjistouj(String1, String2)
char *String1, *String2;
char *csjtojis(String1, String2)
char *String1, *String2;
char *csjtouj(String1, String2)
char *String1, *String2;
char *cujtojis(String1, String2)
char *String1, *String2;
char *cujtosj(String1, String2)
char *String1, *String2;
```

Description

The jistosj, jistouj, sjtojis, sjtouj, ujtojis, and ujtosj subroutines perform string conversion on 8-bit processing codes. The String2 parameter is converted and the converted string is stored in the String1 parameter. The overflow of the String1 parameter is not checked. Also, the String2 parameter must be a valid string. Code validation is not permitted.

The **jistosj** subroutine converts JIS to SJIS. The **jistouj** subroutine converts JIS to UJIS. The **sjtojis** subroutine converts SJIS to JIS. The **sjtouj** subroutine converts SJIS to UJIS. The **ujtojis** subroutine converts UJIS to SJIS.

The **cjistosj**, **cjistouj**, **csjtojis**, **csjtouj**, **cujtojis**, and **cujtosj** macros perform code conversion on 8-bit processing JIS Kanji characters. A character is removed from the *String2* parameter, and its code is converted and stored in the *String1* parameter. The *String1* parameter is returned. The validity of the *String2* parameter is not checked.

The **cjistosj** macro converts from JIS to SJIS. The **cjistouj** macro converts from JIS to UJIS. The **csjtojis** macro converts from SJIS to JIS. The **csjtouj** macro converts from SJIS to UJIS. The **cujtojis** macro converts from UJIS to JIS. The **cujtosj** macro converts from UJIS to SJIS.

Parameters

String1 Stores converted string or code.

String2 Stores string or code to be converted.

Related Information

The "Japanese conv Subroutines" and "Japanese ctype Subroutines" on page 476.

List of String Manipulation Services in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview for Programming in AIX 5L Version 5.2 National Language Support Guide and Reference.

Subroutines, Example Programs, and Libraries in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.*

Japanese conv Subroutines

Purpose

Translates predefined Japanese character classes.

Library

Standard C Library (libc.a)

Syntax

```
#include <ctype.h>
int atojis ( Character)
int Character;

int jistoa (Character)
int Character;

int _atojis (Character)
int Character;

int _jistoa (Character)
int Character;
```

```
int tojupper (Character)
int Character;
int tojlower (Character)
int Character;
int tojupper (Character)
int Character;
int tojlower (Character)
int Character;
int toujis (Character)
int Character;
int kutentojis (Character)
int Character:
int tojhira (Character)
int Character;
int tojkata (Character)
int Character:
```

When running the operating system with Japanese Language Support on your system, the legal value of the *Character* parameter is in the range from 0 to **NLCOLMAX**.

The jistoa subroutine converts an SJIS ASCII equivalent to the corresponding ASCII equivalent. The atojis subroutine converts an ASCII character to the corresponding SJIS equivalent. Other values are returned unchanged.

The _iistoa and _atojis routines are macros that function like the jistoa and atojis subroutines, but are faster and have no error checking function.

The tojlower subroutine converts a SJIS uppercase letter to the corresponding SJIS lowercase letter. The tojupper subroutine converts an SJIS lowercase letter to the corresponding SJIS uppercase letter. All other values are returned unchanged.

The _tojlower and _tojupper routines are macros that function like the tojlower and tojupper subroutines, but are faster and have no error-checking function.

The toujis subroutine sets all parameter bits that are not 16-bit SJIS code to 0.

The **kutentojis** subroutine converts a kuten code to the corresponding SJIS code. The **kutentojis** routine returns 0 if the given kuten code is invalid.

The tojhira subroutine converts an SJIS katakana character to its SJIS hiragana equivalent. Any value that is not an SJIS katakana character is returned unchanged.

The tojkata subroutine converts an SJIS hiragana character to its SJIS katakana equivalent. Any value that is not an SJIS hiragana character is returned unchanged.

The _tojhira and _tojkata subroutines attempt the same conversions without checking for valid input.

For all functions except the toujis subroutine, the out-of-range parameter values are returned without conversion.

Parameters

Character Character to be converted. Pointer Pointer to the escape sequence. CharacterPointer Pointer to a single NLchar data type.

Related Information

The "ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines" on page 168, "conv Subroutines" on page 149, "getc, getchar, fgetc, or getw Subroutine" on page 300, "getwc, fgetwc, or getwchar Subroutine" on page 397, and setlocale subroutine.

List of Character Manipulation Subroutines and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference

Japanese ctype Subroutines

Purpose

Classify characters.

Library

Standard Character Library (libc.a)

Syntax

```
#include <ctype.h>
int isjalpha ( Character)
int Character;
int isjupper (Character)
int Character;
int isjlower (Character)
int Character;
int isjlbytekana (Character)
int Character;
int isjdigit (Character)
int Character;
int isjxdigit (Character)
int Character;
int isjalnum (Character)
int Character;
```

```
int isjspace (Character)
int Character;
int isjpunct (Character)
int Character;
int isjparen (Character)
int Character;
int isparent (Character)
intCharacter;
int isjprint (Character)
int Character;
int isjgraph (Character)
int Character;
int isjis (Character)
int Character;
int isjhira (wc)
wchar t wc;
int isjkanji (wc)
wchar wc;
int isjkata (wc)
wchar t wc;
```

The Japanese ctype subroutines classify character-coded integer values specified in a table. Each of these subroutines returns a nonzero value for True and 0 for False.

Parameters

Character Character to be tested.

Return Values

The **isjprint** and **isjgraph** subroutines return a 0 value for user-defined characters.

Related Information

The "ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines" on page 168, and setlocale subroutine.

List of Character Manipulation Services and Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

kill or killpg Subroutine

Purpose

Sends a signal to a process or to a group of processes.

Library

Standard C Library (libc.a)

Syntax

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

int kill(
    Process,
    Signal)
pid_t Process;
int Signal;

killpg(
    ProcessGroup, Signal)
int ProcessGroup, Signal;
```

Description

The **kill** subroutine sends the signal specified by the *Signal* parameter to the process or group of processes specified by the *Process* parameter.

To send a signal to another process, either the real or the effective user ID of the sending process must match the real or effective user ID of the receiving process, and the calling process must have root user authority.

The processes that have the process IDs of 0 and 1 are special processes and are sometimes referred to here as *proc0* and *proc1*, respectively.

Processes can send signals to themselves.

Note: Sending a signal does not imply that the operation is successful. All signal operations must pass the access checks prescribed by each enforced access control policy on the system.

The following interface is provided for BSD Compatibility:

```
killpg(ProcessGroup, Signal)
int ProcessGroup; Signal;

This interface is equivalent to:
if (ProcessGroup < 0)
{
  errno = ESRCH;
  return (-1);
}
return (kill(-ProcessGroup, Signal));</pre>
```

Parameters

Process Specifies the ID of a process or group of processes.

> If the *Process* parameter is greater than 0, the signal specified by the *Signal* parameter is sent to the process identified by the *Process* parameter.

If the *Process* parameter is 0, the signal specified by the *Signal* parameter is sent to all processes, excluding proc0 and proc1, whose process group ID matches the process group ID of the sender.

If the value of the Process parameter is a negative value other than -1 and if the calling process passes the access checks for the process to be signaled, the signal specified by the Signal parameter is sent to all the processes, excluding proc0 and proc1. If the user ID of the calling process has root user authority, all processes, excluding proc0 and proc1, are signaled.

If the value of the *Process* parameter is a negative value other than -1, the signal specified by the Signal parameter is sent to all processes having a process group ID equal to the absolute value of the *Process* parameter.

If the value of the Process parameter is -1, the signal specified by the Signal parameter is

sent to all processes which the process has permission to send that signal.

Specifies the signal. If the Signal parameter is a null value, error checking is performed but no signal is sent. This parameter is used to check the validity of the *Process* parameter.

ProcessGroup Specifies the process group.

Return Values

Signal

Upon successful completion, the kill subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The kill subroutine is unsuccessful and no signal is sent if one or more of the following are true:

EINVAL The Signal parameter is not a valid signal number.

The Signal parameter specifies the SIGKILL, SIGSTOP, SIGTSTP, or SIGCONT signal, and the Process **EINVAL**

parameter is 1 (proc1).

ESRCH No process can be found corresponding to that specified by the *Process* parameter.

EPERM The real or effective user ID does not match the real or effective user ID of the receiving process, or else

the calling process does not have root user authority.

Related Information

The **getpid**, **getpgrp**, or **getppid** ("getpid, getpgrp, or getppid Subroutine" on page 345) subroutine, setpgid or setpgrp subroutine, sigaction, sigvec, or signal subroutine.

The kill command.

Signal Management in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging *Programs* provides more information about signal management in multi-threaded processes.

kleenup Subroutine

Purpose

Cleans up the run-time environment of a process.

Library

Syntax

```
int kleenup( FileDescriptor, SigIgn, SigKeep)
int FileDescriptor;
int SigIgn[];
int SigKeep[];
```

Description

The **kleenup** subroutine cleans up the run-time environment for a trusted process by:

- Closing unnecessary file descriptors.
- · Resetting the alarm time.
- · Resetting signal handlers.
- · Clearing the value of the real directory read flag described in the ulimit subroutine.
- Resetting the ulimit value, if it is less than a reasonable value (8192).

Parameters

FileDescriptor Specifies a file descriptor. The **kleenup** subroutine closes all file descriptors greater than

or equal to the FileDescriptor parameter.

Siglgn Points to a list of signal numbers. If these are nonnull values, this list is terminated by 0s.

Any signals specified by the *SigIgn* parameter are set to **SIG_IGN**. The handling of all signals not specified by either this list or the *SigKeep* list are set to **SIG_DFL**. Some

signals cannot be reset and are left unchanged.

SigKeep Points to a list of signal numbers. If these are nonnull values, this list is terminated by 0s.

The handling of any signals specified by the SigKeep parameter is left unchanged. The

handling of all signals not specified by either this list or the Siglgn list are set to

SIG_DFL. Some signals cannot be reset and are left unchanged.

Return Values

The **kleenup** subroutine is always successful and returns a value of 0. Errors in closing files are not reported. It is not an error to attempt to modify a signal that the process is not allowed to handle.

Related Information

The ulimit subroutine.

List of Security and Auditing Subroutines and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

knlist Subroutine

Purpose

Translates names to addresses in the running system.

Syntax

#include <nlist.h>

```
int knlist( NList, NumberOfElements, Size)
struct nlist *NList;
int NumberOfElements;
int Size;
```

The knlist subroutine allows a program to examine the list of symbols exported by kernel routines to other kernel modules.

The first field in the **nlist** structure is an input parameter to the **knlist** subroutine. The **n_value** field is modified by the **knlist** subroutine, and all the others remain unchanged. The **nlist** structure consists of the following fields:

char *n name Specifies the name of the symbol whose attributes are to be retrieved.

Indicates the virtual address of the object. This will also be the offset when using segment long n value

descriptor 0 as the extension parameter of the readx or writex subroutines against the

/dev/mem file.

If the name is not found, all fields, other than n name, are set to 0.

The nlist.h file is automatically included by the a.out.h file for compatibility. However, do not include the a.out.h file if you only need the information necessary to use the knlist subroutine. If you do include the **a.out.h** file, follow the **#include** statement with the line:

#undef n name

Notes:

- 1. If both the nlist.h and netdb.h files are to be included, the netdb.h file should be included before the nlist.h file in order to avoid a conflict with the n name structure member. Likewise, if both the a.out.h and **netdb.h** files are to be included, the **netdb.h** file should be included before the **a.out.h** file to avoid a conflict with the n name structure.
- 2. If the netdb.h file and either the nlist.h or syms.h file are included, the n name field will be defined as n. n name. This definition allows you to access the n name field in the nlist or syment structure. If you need to access the n name field in the **netent** structure, undefine the n name field by entering: #undef n name

before accessing the n name field in the netent structure. If you need to access the n name field in a **syment** or **nlist** structure after undefining it, redefine the n name field with:

#define n name n. n name

Parameters

NList Points to an array of nlist structures.

NumberOfElements Specifies the number of structures in the array of **nlist** structures.

Size Specifies the size of each structure.

Return Values

Upon successful completion, knlist returns a value of 0. Otherwise, a value of -1 is returned, and the errno global variable is set to indicate the error.

Error Codes

The knlist subroutine fails when the following is true:

EFAULT Indicates that the NList parameter points outside the limit of the array of nlist structures.

kpidstate Subroutine

Purpose

Returns the status of a process.

Syntax

```
kpidstate (pid)
pid_t pid;
```

Description

The **kpidstate** subroutine returns the state of a process specified by the *pid* parameter. The **kpidstate** subroutine can only be called by a process.

Parameters

pid Specifies the product ID.

Return Values

If the *pid* parameter is not valid, KP_NOTFOUND is returned. If the pid parameter is valid, the following settings in the process state determine what is returned:

SNONE Return KP_NOTFOUND.

SIDL Return KP_INITING.

SZOMB Return KP_EXITING, also if SEXIT in pv_flag.

SSTOP Return KP_STOPPED.

Otherwise the pid is alive and KP_ALIVE is returned.

Error Codes

_lazySetErrorHandler Subroutine

Purpose

Installs an error handler into the lazy loading runtime system for the current process.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/ldr.h>
#include <sys/errno.h>

typedef void (*_handler_t(
    char *_module,
    char *_symbol,
    unsigned int _errVal ))();
handler_t *_lazySetErrorHandler(err_handler)
handler_t *err_handler;
```

This function allows a process to install a custom error handler to be called when a lazy loading reference fails to find the required module or function. This function should only be used when the main program or one of its dependent modules was linked with the -blazy option. To call _lazySetErrorHandler from a module that is not linked with the -blazy option, you must use the -Irtl option. If you use -blazy, you do not need to specify -Irtl.

This function is not thread safe. The calling program should ensure that _lazySetErrorHandler is not called by multiple threads at the same time.

The user-supplied error handler may print its own error message, provide a substitute function to be used in place of the called function, or call the longimp subroutine. To provide a substitute function that will be called instead of the originally referenced function, the error handler should return a pointer to the substitute function. This substitute function will be called by all subsequent calls to the intended function from the same module. If the value returned by the error handler appears to be invalid (for example, a NULL pointer), the default error handler will be used.

Each calling module resolves its lazy references independent of other modules. That is, if module A and B both call foo subroutine in module C, but module C does not export foo subroutine, the error handler will be called once when foo subroutine is called for the first time from A, and once when foo subroutine is called for the first time from B.

The default lazy loading error handler will print a message containing: the name of module that the program required; the name of the symbol being accessed; and the error value generated by the failure. Since the default handler considers a lazy load error to be fatal, the process will exit with a status of 1.

During execution of a program that utilizes lazy loading, there are a few conditions that may cause an error to occur. In all cases the current error handler will be called.

- 1. The referenced module (which is to be loaded upon function invocation) is unavailable or cannot be loaded. The errVal parameter will probably indicate the reason for the error if a system call failed.
- 2. A function is referenced, but the loaded module does not contain a definition for the function. In this case, errVal parameter will be EINVAL.

Some possibilities as to why either of these errors might occur:

- 1. The **LIBPATH** environment variable may contain a set of search paths that cause the application to load the wrong version of a module.
- 2. A module has been changed and no longer provides the same set of symbols that it did when the application was built.
- 3. The **load** subroutine fails due to a lack of resources available to the process.

Parameters

err_handler A pointer to the new error handler function. The new function should accept 3 arguments:

module The name of the referenced module.

symbol The name of the function being called at the time the failure occurred.

The value of **errno** at the time the failure occurred, if a system call used to load the module fails. For other failures, errval may be EINVAL or ENOMEM.

Note that the value of module or symbol may be NULL if the calling module has somehow been corrupted.

If the *err_handler* parameter is NULL, the default error handler is restored.

Return Value

The function returns a pointer to the previous user-supplied error handler, or NULL if the default error handler was in effect.

Related Information

The load ("load Subroutine" on page 618) subroutine.

The Id command.

The Shared Library Overview and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts.

The Shared Library and Lazy Loading in AIX 5L Version 5.2 General Programming Concepts.

13tol or Itol3 Subroutine

Purpose

Converts between 3-byte integers and long integers.

Library

Standard C Library (libc.a)

Syntax

```
void 13tol ( LongPointer, CharacterPointer, Number)
long *LongPointer;
char *CharacterPointer;
int Number;
void 1tol3 (CharacterPointer, LongPointer, Number)
char *CharacterPointer;
long *LongPointer;
int Number;
```

Description

The **I3tol** subroutine converts a list of the number of 3-byte integers specified by the *Number* parameter packed into a character string pointed to by the *CharacterPointer* parameter into a list of long integers pointed to by the *LongPointer* parameter.

The **Itol3** subroutine performs the reverse conversion, from long integers (the *LongPointer* parameter) to 3-byte integers (the *CharacterPointer* parameter).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

Parameters

LongPointerSpecifies the address of a list of long integers.CharacterPointerSpecifies the address of a list of 3-byte integers.NumberSpecifies the number of list elements to convert.

Related Information

The filsys.h file format.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

LAPI_Addr_get Subroutine

Purpose

Retrieves a function address that was previously registered using LAPI_Addr_set.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Addr_get(hndl, addr, addr_hndl)
lapi_handle_t hndl;
void **addr;
int addr_hndl;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_ADDR_GET(hndl, addr, addr_hndl, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: addr
INTEGER addr_hndl
INTEGER ierror
```

Description

Type of call: local address manipulation

Use this subroutine to get the pointer that was previously registered with LAPI and is associated with the index addr_hndl. The value of addr_hndl must be in the range 1 <= addr_hndl < LOC_ADDRTBL_SZ.

Parameters

INPUT

hndl Specifies the LAPI handle.

addr_hndl Specifies the index of the function address to retrieve. You should have previously

registered the address at this index using LAPI_Addr_set. The value of this parameter

must be in the range 1 <= addr_hndl < LOC_ADDRTBL_SZ.

OUTPUT

addr Returns a function address that the user registered with LAPI.

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To retrieve a header handler address that was previously registered using LAPI_Addr_set:

```
addr_hndl = 1;
LAPI_Addr_get(hndl, &addr, addr_hndl);
/* addr now contains the address that was previously registered */
/* using LAPI_Addr_set */
```

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_ADDR_HNDL_RANGE

Indicates that the value of addr_hndl is not in the range 1 <= addr_hndl <

LOC_ADDRTBL_SZ.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_RET_PTR_NULL Indicates that the value of the addr pointer is NULL (in C) or that the value

of addr is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Addr_set, LAPI_Qenv

LAPI_Addr_set Subroutine

Purpose

Registers the address of a function.

Library

Availability Library (liblapi r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Addr_set(hndl, addr, addr_hndl)
lapi_handle_t hndl;
void *addr;
int addr hndl;
```

FORTRAN Syntax

```
include 'lapif.h'
LAPI_ADDR_SET(hndl, addr, addr_hndl, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: addr
INTEGER addr_hndl
INTEGER ierror
```

Description

Type of call: local address manipulation

Use this subroutine to register the address of a function (*addr*). LAPI maintains the function address in an internal table. The function address is indexed at location *addr_hndl*. In subsequent LAPI calls, *addr_hndl* can be used in place of *addr*. The value of *addr_hndl* must be in the range 1 <= *addr_hndl* < LOC_ADDRTBL_SZ.

For active message communication, you can use *addr_hndl* in place of the corresponding header handler address. LAPI only supports this indexed substitution for remote header handler addresses (but not other remote addresses, such as target counters or base data addresses). For these other types of addresses, the actual address value must be passed to the API call.

Parameters

INPUT

hndl Specifies the LAPI handle.

addr Specifies the address of the function handler that the user wants to register with LAPI.

addr_hndl Specifies a user function address that can be passed to LAPI calls in place of a header

handler address. The value of this parameter must be in the range 1 <= addr_hndl <

LOC_ADDRTBL_SZ.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To register a header handler address:

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_ADDR_HNDL_RANGE

Indicates that the value of *addr_hndl* is not in the range 1 <= *addr_hndl* < **LOC ADDRTBL SZ**.

LAPI ERR HNDL INVALID Indicates the

Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI Addr get, LAPI Amsend, LAPI Amsendv, LAPI Qenv, LAPI Xfer

LAPI Address Subroutine

Purpose

Returns an unsigned long value for a specified user address.

Library

Availability Library (liblapi r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Address(my addr, ret addr)
void *my addr;
ulong *ret addr;
```

Note: This subroutine is meant to be used by FORTRAN programs. The C version of LAPI Address is provided for compatibility purposes only.

FORTRAN Syntax

```
include 'lapif.h'
LAPI_ADDRESS(my_addr, ret_addr, ierror)
INTEGER (KIND=any type) :: my addr
INTEGER (KIND=LAPI ADDR TYPE) :: ret addr
INTEGER ierror
```

where:

any_type

Is any FORTRAN datatype. This type declaration has the same meaning as the type void * in C.

Description

Type of call: local address manipulation

Use this subroutine in FORTRAN programs when you need to store specified addresses in an array. In FORTRAN, the concept of address (&) does not exist as it does in C. LAPI Address provides FORTRAN programmers with this function.

Parameters

INPUT

my_addr Specifies the address to convert. The value of this parameter cannot be NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN).

OUTPUT

ret addr Returns the address that is stored in my_addr as an unsigned long for use in LAPI calls.

The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

FORTRAN Examples

To retrieve the address of a variable:

```
! Contains the address of the target counter
integer (KIND=LAPI ADDR TYPE) :: cntr addr
```

```
! Target Counter
type (LAPI CNTR T) :: tgt cntr
! Return code
integer :: ierror
call LAPI ADDRESS(tgt cntr, cntr addr, ierror)
! cntr_addr now contains the address of tgt_cntr
```

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI ERR ORG ADDR NULL

Indicates that the value of my addr is NULL (in C) or LAPI ADDR NULL (in FORTRAN).

LAPI_ERR_TGT_ADDR_NULL

Indicates that the value of ret_addr is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI_Address_init, LAPI_Address_init64

LAPI Address init Subroutine

Purpose

Creates a remote address table.

Library

Availability Library (liblapi r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Address_init(hndl, my addr, add tab)
lapi_handle_t hndl;
void
           *my addr;
            *add tab[];
```

FORTRAN Syntax

```
include 'lapif.h'
LAPI_ADDRESS_INIT(hndl, my_addr, add_tab, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: my addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: add_tab(*)
INTEGER ierror
```

Description

Type of call: collective communication (blocking)

LAPI_Address_init exchanges virtual addresses among tasks of a parallel application. Use this subroutine to create tables of such items as header handlers, target counters, and data buffer addresses.

LAPI_Address_init is a *collective call* over the LAPI handle *hndl*, which fills the table *add_tab* with the virtual address entries that each task supplies. Collective calls must be made in the same order at all participating tasks.

The addresses that are stored in the table *add_tab* are passed in using the *my_addr* parameter. Upon completion of this call, *add_tab[i]* contains the virtual address entry that was provided by task *i*. The array is opaque to the user.

Parameters

INPUT

hndl Specifies the LAPI handle.

my_addr Specifies the entry supplied by each task. The value of this parameter can be NULL (in C)

or LAPI ADDR NULL (in FORTRAN).

OUTPUT

add_tab Specifies the address table containing the addresses that are to be supplied by all tasks.

add_tab is an array of pointers, the size of which is greater than or equal to **NUM_TASKS**. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To collectively transfer target counter addresses for use in a communication API call, in which all nodes are either 32-bit or 64-bit:

For a combination of 32-bit and 64-bit nodes, use LAPI Address init64.

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_COLLECTIVE_PSS

Indicates that a collective call was made while in persistent subsystem

(PSS) mode.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_RET_PTR_NULL Indicates that the value of the add_tab pointer is NULL (in C) or that the

value of add tab is LAPI ADDR NULL (in FORTRAN).

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI Address, LAPI Address init64

LAPI Address init64 Subroutine

Purpose

Creates a 64-bit remote address table.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
int LAPI Address init64(hndl, my addr, add tab)
lapi handle t hndl;
lapi_long_t
             my_addr;
lapi_long_t *add_tab;
```

FORTRAN Syntax

```
include 'lapif.h'
LAPI ADDRESS INIT64(hndl, my addr, add tab, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: my addr
INTEGER (KIND=LAPI_LONG_LONG_TYPE) :: add_tab(*)
INTEGER ierror
```

Description

Type of call: collective communication (blocking)

LAPI Address init64 exchanges virtual addresses among a mixture of 32-bit and 64-bit tasks of a parallel application. Use this subroutine to create 64-bit tables of such items as header handlers, target counters, and data buffer addresses.

LAPI_Address_init64 is a collective call over the LAPI handle hndl, which fills the 64-bit table add_tab with the virtual address entries that each task supplies. Collective calls must be made in the same order at all participating tasks.

The addresses that are stored in the table add tab are passed in using the my addr parameter. Upon completion of this call, add_tab[i] contains the virtual address entry that was provided by task i. The array is opaque to the user.

Parameters

INPUT

hndl Specifies the LAPI handle.

Specifies the address entry that is supplied by each task. The value of this parameter can my_addr

be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN). To ensure 32-bit/64-bit interoperability, it is passed as a lapi long t type in C.

OUTPUT

add tab Specifies the 64-bit address table that contains the 64-bit values supplied by all tasks.

add tab is an array of type lapi long t (in C) or LAPI LONG LONG TYPE (in

FORTRAN). The size of add_tab is greater than or equal to NUM_TASKS. The value of

this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

Specifies a FORTRAN return code. This is always the last parameter. ierror

C Examples

To collectively transfer target counter addresses for use in a communication API call with a mixed task environment (any combination of 32-bit and 64-bit):

```
lapi handle t hndl;
                                       /* the LAPI handle
lapi_long_t addr_tbl[NUM_TASKS]; /* the table for all tasks' addresses */
lapi_long_t tgt_cntr; /* the target counter */
LAPI Address init64(hndl, (lapi long t)&tgt cntr, addr tbl);
/* For communication with task t, use addr tbl[t] as the address */
/* of the target counter. For mixed (32-bit and 64-bit) jobs,
/* use the LAPI Xfer subroutine for communication.
```

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_COLLECTIVE_PSS

Indicates that a collective call was made while in persistent subsystem

(PSS) mode.

LAPI_ERR_HNDL_INVALID Indicates that the *hndl* passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_RET_PTR_NULL Indicates that the value of the add_tab pointer is NULL (in C) or that the

value of add tab is LAPI ADDR NULL (in FORTRAN).

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI Address, LAPI Address init, LAPI Xfer

LAPI Amsend Subroutine

Purpose

Transfers a user message to a remote task, obtaining the target address on the remote task from a user-specified header handler.

Library

Availability Library (**liblapi r.a**)

C Syntax

```
#include <lapi.h>
typedef void (compl hndlr t) (hndl, user info);
                             /* pointer to LAPI context passed in from LAPI Amsend */
lapi handle t *hndl;
               *user_info; /* buffer (user_info) pointer passed in
void
                             /* from header handler (void *(hdr hndlr t))
                                                                                     */
typedef void *(hdr_hndlr_t)(hndl, uhdr, uhdr len, msg len, comp h, user info);
lapi handle t
                *hndl;
                              /* pointer to LAPI context passed in from LAPI Amsend */
                              /* uhdr passed in from LAPI Amsend
void
                *uhdr;
uint
                *uhdr len;
                              /* uhdr len passed in from LAPI Amsend
                                                                                      */
                *msg_len;
ulong
                              /* udata len passed in fom LAPI Amsend
compl_hndlr_t **comp_h;
                              /* function address of completion handler
                              /* (void (compl_hndlr_t)) that needs to be filled
                              /* out by this header handler function.
               **user_info; /* pointer to the parameter to be passed
void
                              /* in to the completion handler
\textbf{int LAPI\_Amsend(} \textit{hndl, } \textit{tgt, } \textit{hdr\_hdl, } \textit{uhdr, } \textit{uhdr\_len, } \textit{udata\_len, } \\
                tgt_cntr, org_cntr, cmpl_cntr)
lapi handle t hndl;
uint
               tgt;
              *hdr hdl;
void
void
              *uhdr;
uint
              uhdr len;
void
              *udata;
ulong
              udata len;
lapi_cntr_t
             *tgt_cntr;
lapi_cntr_t
              *org_cntr;
lapi cntr t
              *cmpl cntr;
FORTRAN Syntax
include 'lapif.h'
INTEGER SUBROUTINE COMPL H (hndl, user info)
INTEGER hndl
INTEGER user info
INTEGER FUNCTION HDR_HDL (hndl, uhdr, uhdr len, msg len, comp h, user info)
INTEGER hndl
INTEGER uhdr
INTEGER uhdr len
INTEGER (KIND=LAPI LONG TYPE) :: msg len
EXTERNAL INTEGER FUNCTION comp h
TYPE (LAPI_ADDR_T) :: user_info
LAPI_AMSEND(hndl, tgt, hdr_hdl, uhdr, uhdr_len, udata, udata_len,
            tgt_cntr, org_cntr, cmpl_cntr, ierror)
INTEGER hndl
INTEGER tgt
EXTERNAL INTEGER FUNCTION hdr_hdl
INTEGER uhdr
INTEGER uhdr len
TYPE (LAPI ADDR T) :: udata
INTEGER (KIND=LAPI_LONG_TYPE) :: udata_len
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror
```

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to transfer data to a target task, where it is desirable to run a handler on the target task before message delivery begins or after delivery completes. LAPI_Amsend allows the user to provide a header handler and optional completion handler. The header handler is used to specify the target buffer address for writing the data, eliminating the need to know the address on the origin task when the subroutine is called.

User data (uhdr and udata) are sent to the target task. Once these buffers are no longer needed on the origin task, the origin counter is incremented, which indicates the availability of origin buffers for modification. Using the LAPI_Xfer call with the LAPI_AM_XFER type provides the same type of transfer, with the option of using a send completion handler instead of the origin counter to specify buffer availability.

Upon arrival of the first data packet at the target, the user's header handler is invoked. Note that a header handler must be supplied by the user because it returns the base address of the buffer in which LAPI will write the data sent from the origin task (udata). See RSCT for AIX 5L: LAPI Programming Guide for an optimization exception to this requirement that a buffer address be supplied to LAPI for single-packet messages.

The header handler also provides additional information to LAPI about the message delivery, such as the completion handler. LAPI Amsend and similar calls (such as LAPI Amsendv and corresponding LAPI Xfer transfers) also allow the user to specify their own message header information, which is available to the header handler. The user may also specify a completion handler parameter from within the header handler. LAPI will pass the information to the completion handler at execution.

Note that the header handler is run inline by the thread running the LAPI dispatcher. For this reason, the header handler must be non-blocking because no other progress on messages will be made until it returns. It is also suggested that execution of the header handler be simple and quick. The completion handler, on the other hand, is normally enqueued for execution by a separate thread. It is possible to request that the completion handler be run inline. See RSCT for AIX 5L: LAPI Programming Guide for more information on inline completion handlers.

If a completion handler was not specified (that is, set to LAPI_ADDR_NULL in FORTRAN or its pointer set to NULL in C), the arrival of the final packet causes LAPI to increment the target counter on the remote task and send an internal message back to the origin task. The message causes the completion counter (if it is not NULL in C or LAPI ADDR NULL in FORTRAN) to increment on the origin task.

If a completion handler was specified, the above steps take place after the completion handler returns. To quarantee that the completion handler has executed on the target, you must wait on the completion counter. See RSCT for AIX 5L: LAPI Programming Guide for a time-sequence diagram of events in a LAPI Amsend call.

User details

As mentioned above, the user must supply the address of a header handler to be executed on the target upon arrival of the first data packet. The signature of the header handler is as follows:

```
void *hdr hndlr(lapi handle t *hndl, void *uhdr, uint *uhdr len, ulong *msg len,
                compl_hndlr_t **cmpl_hndlr, void **user_info);
```

The value returned by the header handler is interpreted by LAPI as an address for writing the user data (udata) that was passed to the LAPI_Amsend call. The uhdr and uhdr_len parameters are passed by LAPI into the header handler and contain the information passed by the user to the corresponding parameters of the LAPI Amsend call.

Use of LAPI Addr set

Remote addresses are commonly exchanged by issuing a collective LAPI_Address_init call within a few steps of initializing LAPI. LAPI also provides the LAPI_Addr_set mechanism, whereby users can register one or more header handler addresses in a table, associating an index value with each address. This index can then be passed to LAPI Amsend instead of an actual address. On the target side, LAPI will use the index to get the header handler address. Note that, if all tasks use the same index for their header handler, the initial collective communication can be avoided. Each task simply registers its own header handler address using the well-known index. Then, on any LAPI_Amsend calls, the reserved index can be passed to the header handler address parameter.

Role of the header handler

The user optionally returns the address of a completion handler function through the cmpl_hndlr parameter and a completion handler parameter through the user_info parameter. The address passed through the user info parameter can refer to memory containing a datatype defined by the user and then cast to the appropriate type from within the completion handler if desired.

The signature for a user completion handler is as follows: typedef void (compl hndlr t)(lapi handle t *hndl, void *completion param);

The argument returned by reference through the user info member of the user's header handler will be passed to the completion_param argument of the user's completion handler. See the C Examples for an example of setting the completion handler and parameter in the header handler.

As mentioned above, the value returned by the header handler must be an address for writing the user data sent from the origin task. There is one exception to this rule. In the case of a single-packet message, LAPI passes the address of the packet in the receive FIFO, allowing the entire message to be consumed within the header handler. In this case, the header handler should return NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) so that LAPI does not copy the message to a target buffer. See RSCT for AIX 5L: LAPI Programming Guide for more information (including a sample header handler that uses this method for fast retrieval of a single-packet message).

Passing additional information through lapi_return_info_t

LAPI allows additional information to be passed to and returned from the header handler by passing a pointer to lapi return info t through the msg len argument. On return from a header handler that is invoked by a call to LAPI Amsend, the ret flags member of lapi return info t can contain one of these values: LAPI_NORMAL (the default), LAPI_SEND_REPLY (to run the completion handler inline), or LAPI LOCAL STATE (no reply is sent). The dgsp handle member of lapi return info t should not be used in conjunction with LAPI Amsend.

For a complete description of the lapi return info t type, see RSCT for AIX 5L: LAPI Programming Guide.

Inline execution of completion handlers

Under normal operation, LAPI uses a separate thread for executing user completion handlers, After the final packet arrives, completion handler pointers are placed in a queue to be handled by this thread. For performance reasons, the user may request that a given completion handler be run inline instead of being placed on this queue behind other completion handlers. This mechanism gives users a greater degree of control in prioritizing completion handler execution for performance-critical messages.

LAPI places no restrictions on completion handlers that are run "normally" (that is, by the completion handler thread). Inline completion handlers should be short and should not block, because no progress can be made while the main thread is executing the handler. The user must use caution with inline

completion handlers so that LAPI's internal queues do not fill up while waiting for the handler to complete. I/O operations must not be performed with an inline completion handler.

Parameters

INPUT

hndl Specifies the LAPI handle.

tgt Specifies the task ID of the target task. The value of this parameter must be in the range 0

 $<= tgt < NUM_TASKS.$

hdr_hdl Specifies the pointer to the remote header handler function to be invoked at the target.

The value of this parameter can take an address handle that has already been registered

using LAPI_Addr_set. The value of this parameter cannot be NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN).

uhdr Specifies the pointer to the user header data. This data will be passed to the user header

handler on the target. If uhdr_len is 0, The value of this parameter can be NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN).

uhdr_len Specifies the length of the user's header. The value of this parameter must be a multiple

of the processor's doubleword size in the range 0 <= uhdr_len <= MAX_UHDR_SZ.

udata Specifies the pointer to the user data. If udata_len is **0**, The value of this parameter can

be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

udata_len Specifies the length of the user data in bytes. The value of this parameter must be in the

range 0 <= udata_len <= the value of LAPI constant LAPI_MAX_MSG_SZ.

INPUT/OUTPUT

tgt_cntr Specifies the target counter address. The target counter is incremented after the

completion handler (if specified) completes or after the completion of data transfer. If the value of this parameter is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the target

counter is not updated.

org_cntr Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin

counter is incremented after data is copied out of the origin address (in C) or the origin (in

FORTRAN). If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN), the origin counter is not updated.

cmpl_cntr Specifies the counter at the origin that signifies completion of the completion handler. It is

updated once the completion handler completes. If no completion handler is specified, the counter is incremented at the completion of message delivery. If the value of this

parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the completion counter is

not updated.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To send an active message and then wait on the completion counter:

```
lapi handle t hndl;
                                        /* the LAPI handle
                                       /* the LAPI task ID
            task id;
int
            num tasks;
                                       /* the total number of tasks
int
void
            *hdr_hndlr_list[NUM_TASKS]; /* the table of remote header handlers */
                          /* the communication partner
int
            buddy;
                                      /* the completion counter
lapi cntr t cmpl cntr;
            data buffer[DATA LEN]; /* the data to transfer
int.
/* retrieve header handler addresses */
LAPI Address init(hndl, (void *)&hdr hndlr, hdr hndlr list);
** up to this point, all instructions have executed on all
** tasks. we now begin differentiating tasks.
*/
if ( sender ) {
                                /* origin task */
   /* initialize data buffer, cmpl cntr, etc. */
    /* synchronize before starting data transfer */
   LAPI Gfence(hndl);
   LAPI Amsend(hndl, buddy, (void *)hdr hndlr list[buddy], NULL,
               0,&(data buffer[0]),DATA LEN*(sizeof(int)),
               NULL, NULL, cmpl cntr);
    /* Wait on completion counter before continuing. Completion
   /* counter will update when message completes at target.
                                                     /* receiver */
} else {
   /* to match the origin's synchronization before data transfer */
   LAPI Gfence(hnd1);
}
```

For a complete program listing, see RSCT for AIX 5L: LAPI Programming Guide. Sample code illustrating the LAPI_Amsend call can be found in the LAPI sample files. See RSCT for AIX 5L: LAPI Programming Guide for more information about the sample programs that are shipped with LAPI.

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

Indicates that the value of udata_len is greater than the value of LAPI LAPI_ERR_DATA_LEN

constant LAPI_MAX_MSG_SZ.

LAPI_ERR_HDR_HNDLR_NULL

Indicates that the value of the hdr_hdl passed in is NULL (in C) or LAPI ADDR NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_ADDR_NULL

Indicates that the value of the *udata* parameter passed in is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), but the value of *udata_len* is

greater than 0.

LAPI_ERR_TGT Indicates that the *tgt* passed in is outside the range of tasks defined in the

job.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask()

was called.

LAPI_ERR_UHDR_LEN Indicates that the *uhdr_len* value passed in is greater than

MAX_UHDR_SZ or is not a multiple of the processor's doubleword size.

LAPI_ERR_UHDR_NULL Indicates that the *uhdr* passed in is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN), but *uhdr_len* is not **0**.

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Addr_get, LAPI_Addr_set, LAPI_Getcntr, LAPI_MsgpoII, LAPI_Qenv, LAPI_Setcntr, LAPI_Waitcntr, LAPI_Xfer

LAPI Amsendy Subroutine

Purpose

Transfers a user vector to a remote task, obtaining the target address on the remote task from a user-specified header handler.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
typedef void (compl_hndlr_t) (hndl, user_info);
                          /* the LAPI handle passed in from LAPI Amsendv */
lapi_handle_t *hnd\overline{l};
void
               *user info; /* the buffer (user info) pointer passed in
                            /* from vhdr hndlr (void *(vhdr hndlr t))
typedef lapi_vec_t *(vhdr hndlr t) (hndl, uhdr, uhdr len, len vec, comp h, uinfo);
lapi handle t *hndl;
                            /* pointer to the LAPI handle passed in from LAPI Amsendv */
void
                            /* uhdr passed in from LAPI Amsendv
               *uhdr len; /* uhdr len passed in from LAPI Amsendv
uint
               *len_vec[]; /* vector of lengths passed in LAPI Amsendv
ulong
compl_hndlr_t **comp_h;
                            /* function address of completion handler
                            /* (void (compl hndlr t)) that needs to be
                            /* filled out by this header handler function
              **user_info; /* pointer to the parameter to be passed
void
                            /* in to the completion handler
int LAPI_Amsendv(hndl, tgt, hdr_hdl, uhdr, uhdr_len, org_vec,
                 tgt_cntr, org_cntr, cmpl_cntr);
```

```
lapi handle t hndl;
uint
               tgt;
               *hdr hdl;
void
              *uhdr;
void
              uhdr len;
uint
lapi vec t
              *org vec;
lapi cntr t
              *tgt cntr;
lapi_cntr_t
              *org_cntr;
lapi_cntr_t
              *cmpl_cntr;
```

FORTRAN Syntax

```
include 'lapif.h'
INTEGER SUBROUTINE COMPL_H (hndl, user info)
INTEGER hndl
INTEGER user info(*)
INTEGER FUNCTION VHDR_HDL (hndl, uhdr, uhdr len, len vec, comp h, user info)
INTEGER hndl
INTEGER uhdr
INTEGER uhdr len
INTEGER (KIND=LAPI LONG TYPE) :: len vec
EXTERNAL INTEGER FUNCTION comp h
TYPE (LAPI_ADDR_T) :: user info
LAPI AMSENDV (hndl, tgt, hdr hdl, uhdr, uhdr len, org vec,
             tgt cntr, org cntr, cmpl cntr, ierror)
INTEGER hndl
INTEGER tgt
EXTERNAL INTEGER FUNCTION hdr hdl
INTEGER uhdr
INTEGER uhdr len
TYPE (LAPI_VEC_T) :: org vec
INTEGER (KIND=LAPI ADDR TYPE) :: tgt cntr
TYPE (LAPI CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror
```

Description

Type of call: point-to-point communication (non-blocking)

LAPI_Amsendv is the vector-based version of the **LAPI_Amsend** call. You can use it to specify multi-dimensional and non-contiguous descriptions of the data to transfer. Whereas regular LAPI calls allow the specification of a single data buffer address and length, the vector versions allow the specification of a vector of address and length combinations. Additional information is allowed in the data description on the origin task and the target task.

Use this subroutine to transfer a vector of data to a target task, when you want a handler to run on the target task before message delivery begins or after message delivery completes.

To use **LAPI_Amsendv**, you must provide a header handler, which returns the address of the target vector description that LAPI uses to write the data that is described by the origin vector. The header handler is used to specify the address of the vector description for writing the data, which eliminates the need to know the description on the origin task when the subroutine is called. The header handler is called upon arrival of the first data packet at the target.

Optionally, you can also provide a completion handler. The header handler provides additional information to LAPI about the message delivery, such as the completion handler. You can also specify a completion handler parameter from within the header handler. LAPI passes the information to the completion handler at execution.

With the exception of the address that is returned by the completion handler, the use of counters, header handlers, and completion handlers in **LAPI_Amsendv** is identical to that of **LAPI_Amsend**. In both cases, the user header handler returns information that LAPI uses for writing at the target. See **LAPI_Amsend** for more information. This section presents information that is specific to the vector version of the call (**LAPI Amsendv**).

LAPI vectors are structures of type lapi_vec_t, defined as follows:

```
typedef struct {
    lapi_vectype_t vec_type;
    uint num_vecs;
    void **info;
    ulong *len;
} lapi vec t;
```

vec_type is an enumeration that describes the type of vector transfer, which can be: LAPI_GEN_GENERIC, LAPI_GEN_IOVECTOR, or LAPI_GEN_STRIDED_XFER.

For transfers of type LAPI_GEN_GENERIC and LAPI_GEN_IOVECTOR, the fields are used as follows:

num_vecs indicates the number of data vectors to transfer. Each data vector is defined by a base

address and data length.

info is the array of addresses.

len is the array of data lengths.

For example, consider the following vector description:

```
vec_type = LAPI_GEN_IOVECTOR
num_vecs = 3
info = {addr_0, addr_1, addr_2}
len = {len 0, len 1, len 2}
```

On the origin side, this example would tell LAPI to read len_0 bytes from addr_0, len_1 bytes from addr_1, and len_2 bytes from addr_2. As a target vector, this example would tell LAPI to write len_0 bytes to addr_0, len_1 bytes to addr_1, and len_2 bytes to addr_2.

Recall that vector transfers require an origin and target vector. For **LAPI_Amsendv** calls, the origin vector is passed to the API call on the origin task. The address of the target vector is returned by the header handler.

For transfers of type **LAPI_GEN_GENERIC**, the target vector description must also have type **LAPI_GEN_GENERIC**. The contents of the *info* and *len* arrays are unrestricted in the generic case; the number of vectors and the length of vectors on the origin and target do not need to match. In this case, LAPI transfers a given number of bytes in noncontiguous buffers specified by the origin vector to a set of noncontiguous buffers specified by the target vector.

If the sum of target vector data lengths (say TGT_LEN) is less than the sum of origin vector data lengths (say ORG_LEN), only the first TGT_LEN bytes from the origin buffers are transferred and the remaining bytes are discarded. If TGT_LEN is greater than ORG_LEN, all ORG_LEN bytes are transferred. Consider the following example:

```
num_vecs = 4;
info = {tgtaddr_0, tgtaddr_1, tgtaddr_2, tgtaddr_3};
len = {12, 2, 4, 2}
```

LAPI copies data as follows:

- 1. 5 bytes from orgaddr_0 to tgtaddr_0 (leaves 7 bytes of space at a 5-byte offset from tgtaddr_0)
- 2. 7 bytes from orgaddr_1 to remaining space in tgtaddr_0 (leaves 3 bytes of data to transfer from orgaddr_1)
- 3. 2 bytes from orgaddr_1 to tgtaddr_1 (leaves 1 byte to transfer from orgaddr_1)
- 4. 1 byte from orgaddr_1 followed by 3 bytes from orgaddr_2 to tgt_addr_2 (leaves 3 bytes to transfer from orgaddr_2)
- 5. 2 bytes from orgaddr 2 to tgtaddr 3

LAPI will copy data from the origin until the space described by the target is filled. For example:

```
Origin_vector: {
    num_vecs = 1;
    info = {orgaddr_0};
    len = {20}
}

Target_vector: {
    num_vecs = 2;
    info = {tgtaddr_0, tgtaddr_1};
    len = {5, 10}
}
```

LAPI will copy 5 bytes from orgaddr_0 to tgtaddr_0 and the next 10 bytes from orgaddr_0 to tgtaddr_1. The remaining 5 bytes from orgaddr_0 will not be copied.

For transfers of type **LAPI_GEN_IOVECTOR**, the lengths of the vectors must match and the target vector description must match the origin vector description. More specifically, the target vector description must:

- also have type LAPI_GEN_IOVECTOR
- have the same num_vecs as the origin vector
- initialize the info array with *num_vecs* addresses in the target address space. For LAPI vectors origin vector and target vector described similarly to the example above, data is copied as follows:
 - 1. transfer origin_vector.len[0] bytes from the address at origin_vector.info[0] to the address at target vector.info[0]
 - 2. transfer origin_vector.len[1] bytes from the address at origin_vector.info[1] to the address at target_vector.info[1]
 - 3. transfer origin_vector.len[n] bytes from the address at origin_vector.info[n] to the address at target_vector.info[n], for n = 2 to $n = [num_vecs-3]$
 - 4. transfer origin_vector.len[num_vecs-2] bytes from the address at origin_vector.info[num_vecs-2] to the address at target_vector.info[num_vecs-2]
 - 5. copy origin_vector.len[num_vecs-1] bytes from the address at origin_vector.info[num_vecs-1] to the address at target_vector.info[num_vecs-1]

Strided vector transfers

For transfers of type **LAPI_GEN_STRIDED_XFER**, the target vector description must match the origin vector description. Rather than specifying the set of address and length pairs, the *info* array of the origin and target vectors is used to specify a data block "template", consisting of a base address, block size and stride. LAPI thus expects the *info* array to contain three integers. The first integer contains the base address, the second integer contains the block size to copy, and the third integer contains the byte stride. In this case, *num_vecs* indicates the number of blocks of data that LAPI should copy, where the first block

begins at the base address. The number of bytes to copy in each block is given by the block size and the starting address for all but the first block is given by previous address + stride. The total amount of data to be copied will be *num_vecs*block_size*. Consider the following example:

```
Origin vector {
   num vecs = 3;
   info = {orgaddr, 5, 8}
```

Based on this description, LAPI will transfer 5 bytes from orgaddr, 5 bytes from orgaddr+8 and 5 bytes from orgaddr+16.

Call details

As mentioned above, counter and handler behavior in LAPI Amsendy is nearly identical to that of LAPI Amsend. A short summary of that behavior is provided here. See the LAPI Amsend description for full details.

This is a non-blocking call. The calling task cannot change the *uhdr* (origin header) and *org vec* data until completion at the origin is signaled by the org_cntr being incremented. The calling task cannot assume that the org vec structure can be changed before the origin counter is incremented. The structure (of type lapi vec t) that is returned by the header handler cannot be modified before the target counter has been incremented. Also, if a completion handler is specified, it may execute asynchronously, and can only be assumed to have completed after the target counter increments (on the target) or the completion counter increments (at the origin).

The length of the user-specified header (uhdr_len) is constrained by the implementation-specified maximum value MAX_UHDR_SZ. uhdr_len must be a multiple of the processor's doubleword size. To get the best bandwidth, *uhdr_len* should be as small as possible.

If the following requirement is not met, an error condition occurs:

· If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the target side, the contents of the target buffer are undefined after the operation.

Parameters

hndl	Specifies the LAPI handle.
tgt	Specifies the task ID of the target task. The value of this parameter must be in the range $0 <= tgt < NUM_TASKS$.
hdr_hdl	Points to the remote header handler function to be invoked at the target. The value of this parameter can take an address handle that had been previously registered using the LAPI_Addr_set/LAPI_Addr_get mechanism. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
uhdr	Specifies the pointer to the local header (parameter list) that is passed to the handler function. If <i>uhdr_len</i> is 0 , The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
uhdr_len	Specifies the length of the user's header. The value of this parameter must be a multiple of the processor's doubleword size in the range 0 <= <i>uhdr_len</i> <= MAX_UHDR_SZ .
org_vec	Points to the origin vector.

INPUT/OUTPUT

tgt_cntr

Specifies the target counter address. The target counter is incremented after the completion handler (if specified) completes or after the completion of data transfer. If the value of this parameter is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the target counter is not updated.

org_cntr

Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data is copied out of the origin address (in C) or the origin (in FORTRAN). If the value of this parameter is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the origin counter is not updated.

cmpl_cntr

Specifies the counter at the origin that signifies completion of the completion handler. It is updated once the completion handler completes. If no completion handler is specified, the counter is incremented at the completion of message delivery. If the value of this parameter is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the completion counter is not updated.

OUTPUT

ierror

Specifies a FORTRAN return code. This is always the last parameter.

C Examples

1. To send a LAPI_GEN_IOVECTOR using active messages:

```
/* header handler routine to execute on target task */
lapi vec t *hdr hndlr(lapi handle t *handle, void *uhdr, uint *uhdr len,
                      ulong *len vec[], compl hndlr t **completion handler,
        void **user info)
     /* set completion handler pointer and other info */
     /* set up the vector to return to LAPI
     /* for a LAPI_GEN_IOVECTOR: num_vecs, vec_type, and len must all have */
     /* the same values as the origin vector. The info array should
     /* contain the buffer addresses for LAPI to write the data
     vec->num vecs = NUM VECS;
     vec->vec type = LAPI GEN IOVECTOR;
                  = (unsigned long *)malloc(NUM VECS*sizeof(unsigned long));
     vec->info
                  = (void **) malloc(NUM VECS*sizeof(void *));
     for( i=0; i < NUM VECS; i++ ) {</pre>
         vec->info[i] = (void *) &data buffer[i];
         vec->len[i] = (unsigned long)(sizeof(int));
     }
     return vec;
{
                 *hdr hndlr list[NUM TASKS]; /* table of remote header handlers */
     lapi vec t *vec;
                                             /* data for data transfer
     vec->num vecs = NUM VECS;
     vec->vec type = LAPI GEN IOVECTOR;
                = (unsigned long *) malloc(NUM VECS*sizeof(unsigned long));
                   = (void **) malloc(NUM_VECS*sizeof(void *));
     /* each vec->info[i] gets a base address
     /* each vec->len[i] gets the number of bytes to transfer from vec->info[i] */
```

```
LAPI_Amsendv(hndl, tgt, (void *) hdr_hdl_list[buddy], NULL, 0, vec,
                  tgt cntr, org cntr, cmpl cntr);
     /* data will be copied as follows:
     /* len[0] bytes of data starting from address info[0] */
     /* len[1] bytes of data starting from address info[1] */
     /* len[NUM VECS-1] bytes of data starting from address info[NUM VECS-1] */
}
```

The above example could also illustrate the LAPI_GEN_GENERIC type, with the following modifications:

- Both vectors would need LAPI_GEN_GENERIC as the vec_type.
- · There are no restrictions on symmetry of number of vectors and lengths between the origin and target sides.
- 2. To send a LAPI_STRIDED_VECTOR using active messages:

```
/* header handler routine to execute on target task */
lapi vec t *hdr hndlr(lapi handle t *handle, void *uhdr, uint *uhdr len,
                       ulong *len_vec[], compl_hndlr_t **completion_handler,
        void **user_info)
{
     int block size;
                                /* block size */
                                /* stride */
     int data size;
     vec->num vecs = NUM VECS;
                                     /* NUM VECS = number of vectors to transfer */
                                     /* must match that of the origin vector */
     vec->vec type = LAPI GEN STRIDED XFER;
                                                       /* same as origin vector */
     /* see comments in origin vector setup for a description of how data
     /* will be copied based on these settings.
                                                                                    */
     vec->info[0] = buffer_address; /* starting address for data copy */
vec->info[1] = block_size; /* bytes of data to copy */
vec->info[2] = stride; /* distance from copy block to copy block */
     return vec;
}
{
                                                            /* data for data transfer */
      lapi vec t *vec;
      vec->num vecs = NUM VECS;
                                         /* NUM VECS = number of vectors to transfer */
                                         /* must match that of the target vector
      vec->vec type = LAPI GEN STRIDED XFER;
                                                        /* same as target vector
      vec->info[0] = buffer address;
                                           /* starting address for data copy
      vec->info[1] = block_size;
                                          /* bytes of data to copy
      vec->info[2] = stride;
                                          /* distance from copy block to copy block */
      /* data will be copied as follows:
      /* block size bytes will be copied from buffer address
      /* block_size bytes will be copied from buffer_address+stride
      /* block size bytes will be copied from buffer address+(2*stride)
```

For complete examples, see the sample programs shipped with LAPI.

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HDR_HNDLR_NULL

Indicates that the *hdr_hdl* passed in is NULL (in C) or **LAPI_ADDR_NULL**

(in FORTRAN).

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_ORG_EXTENT Indicates that the *org_vec*'s extent (stride * *num_vecs*) is greater than the

value of LAPI constant LAPI MAX MSG SZ.

LAPI_ERR_ORG_STRIDE Indicates that the *org_vec* stride is less than block.

LAPI_ERR_ORG_VEC_ADDR

Indicates that the org_vec->info[i] is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN), but its length (org_vec->len[i]) is not **0**.

LAPI_ERR_ORG_VEC_LEN Indicates that the sum of *org_vec->len* is greater than the value of LAPI

 $constant \ \, \textbf{LAPI_MAX_MSG_SZ}.$

LAPI_ERR_ORG_VEC_NULL Indicates that org_vec is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN).

LAPI_ERR_ORG_VEC_TYPE Indicates that the *org_vec->vec_type* is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL

Indicates that the strided vector address org_vec->info[0] is NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT Indicates that the *tgt* passed in is outside the range of tasks defined in the

job.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask()

was called.

LAPI_ERR_UHDR_LEN Indicates that the *uhdr_len* value passed in is greater than

MAX_UHDR_SZ or is not a multiple of the processor's doubleword size.

LAPI_ERR_UHDR_NULL Indicates that the *uhdr* passed in is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN), but uhdr_len is not 0.

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Address_init, LAPI_Addr_get, LAPI_Addr_set, LAPI_Amsend, LAPI_Getcntr, LAPI_Getv, LAPI_Putv, LAPI_Qenv, LAPI_Waitcntr, LAPI_Xfer

LAPI_Fence Subroutine

Purpose

Enforces order on LAPI calls.

Library

Availability Library (liblapi r.a)

C Syntax

#include <lapi.h>

int LAPI_Fence(hndl)
lapi_handle_t hndl;

FORTRAN Syntax

include 'lapif.h'

LAPI_FENCE(hndl, ierror)
INTEGER hndl
INTEGER ierror

Description

Type of call: Local data synchronization (may require progress on the remote task)

Use this subroutine to enforce order on LAPI calls. If a task calls **LAPI_Fence**, all the LAPI operations that were initiated by that task, before the fence using the LAPI context *hndl*, are guaranteed to complete at the target tasks. This occurs before any of its communication operations using *hndl*, initiated after the **LAPI_Fence**, start transmission of data. This is a data fence which means that the data movement is complete. This is not an operation fence which would need to include active message completion handlers completing on the target.

LAPI_Fence may require internal protocol processing on the remote side to complete the fence request.

Parameters

INPUT

hndl Specifies the LAPI handle.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To establish a data barrier in a single task:

```
lapi handle t hndl; /* the LAPI handle */
/* API communication call 1 */
/* API communication call 2 */
/* API communication call n */
LAPI Fence(hnd1);
/* all data movement from above communication calls has completed by this point */
/st any completion handlers from active message calls could still be running.
```

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the *hndl* passed in is not valid (not initialized or in

terminated state).

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI_Amsend, LAPI_Gfence

LAPI Get Subroutine

Purpose

Copies data from a remote task to a local task.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Get(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr)
lapi_handle_t hndl;
uint
             tgt;
ulong
             len;
void
            *tgt_addr;
void
            *org addr;
lapi_cntr_t *tgt cntr;
lapi_cntr_t *org_cntr;
```

FORTRAN Syntax

```
include 'lapif.h'
LAPI_GET(hndl, tgt, len, tgt addr, org addr, tgt cntr, org cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_LONG_TYPE) :: len
```

```
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: org_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror
```

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to transfer data from a remote (target) task to a local (origin) task. Note that in this case the origin task is actually the *receiver* of the data. This difference in transfer type makes the counter behavior slightly different than in the normal case of origin sending to target.

The origin buffer will still increment on the origin task upon availability of the origin buffer. But in this case, the origin buffer becomes available once the transfer of data is complete. Similarly, the target counter will increment once the target buffer is available. Target buffer availability in this case refers to LAPI no longer needing to access the data in the buffer.

This is a non-blocking call. The caller *cannot* assume that data transfer has completed upon the return of the function. Instead, counters should be used to ensure correct buffer addresses as defined above.

Note that a zero-byte message does not transfer data, but it does have the same semantic with respect to counters as that of any other message.

Parameters

INPUT

hndl Specifies the LAPI handle.

tgt Specifies the task ID of the target task that is the source of the data. The value of this

parameter must be in the range **0** <= *tgt* < **NUM_TASKS**.

len Specifies the number of bytes of data to be copied. This parameter must be in the range 0

<= len <= the value of LAPI constant LAPI MAX MSG SZ.

tgt addr Specifies the target buffer address of the data source. If len is 0, The value of this

parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

INPUT/OUTPUT

tgt_cntr Specifies the target counter address. The target counter is incremented once the data

buffer on the target can be modified. If the value of this parameter is NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.

org_cntr Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin

counter is incremented after data arrives at the origin. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

OUTPUT

org_addr Specifies the local buffer address into which the received data is copied. If len is 0, The

value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_DATA_LEN Indicates that the value of *udata_len* is greater than the value of LAPI

constant LAPI_MAX_MSG_SZ.

LAPI ERR HNDL INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_ORG_ADDR_NULL

Indicates that the *org_addr* passed in is NULL (in C) or

LAPI ADDR NULL (in FORTRAN), but *len* is greater than **0**.

LAPI_ERR_TGT Indicates that the *tgt* passed in is outside the range of tasks defined in the

job.

LAPI_ERR_TGT_ADDR_NULL

Indicates that the tgt_addr passed in is NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN), but *len* is greater than **0**.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask()

was called.

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Address_init, LAPI_Getcntr, LAPI_Put, LAPI_Qenv, LAPI_Waitcntr, LAPI_Xfer

LAPI_Getcntr Subroutine

Purpose

Gets the integer value of a specified LAPI counter.

Library

Availability Library (liblapi r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Getcntr(hndl, cntr, val)
lapi_handle_t hndl;
lapi_cntr_t *cntr;
int *val;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_GETCNTR(hndl, cntr, val, ierror)
INTEGER hndl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER val
INTEGER ierror
```

Description

Type of call: Local counter manipulation

This subroutine gets the integer value of cntr. It is used to check progress on hndl.

Parameters

INPUT

hndl Specifies the LAPI handle.

cntr Specifies the address of the counter. The value of this parameter cannot be NULL (in C)

or LAPI_ADDR_NULL (in FORTRAN).

OUTPUT

val Returns the integer value of the counter *cntr*. The value of this parameter cannot be NULL

(in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

```
{
    lapi_cntr_t cntr;
    int val;

/* cntr is initialized */

/* processing/communication takes place */

LAPI_Getcntr(hndl, &cntr, &val)

/* val now contains the current value of cntr */
```

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL Indicates that the *cntr* pointer is NULL (in C) or that the value of *cntr* is

LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_RET_PTR_NULL Indicates that the value of the val pointer is NULL (in C) or that the value

of val is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Amsend, LAPI_Amsendv, LAPI_Get, LAPI_Getv, LAPI_Put, LAPI_Putv, LAPI_Rmw, LAPI_Setcntr, LAPI_Waitcntr, LAPI_Xfer

LAPI_Getv Subroutine

Purpose

Copies vectors of data from a remote task to a local task.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Getv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr)
lapi handle t hndl;
uint
           tgt;
lapi_vec_t *tgt_vec;
lapi_vec_t *org_vec;
lapi_cntr_t *tgt_cntr;
lapi cntr t *org cntr;
typedef struct {
   lapi vectype t vec type; /* operation code
   uint     num_vecs; /* number of vectors
                                                   */
                 **info; /* vector of information */
   void
   ulong
                 *len;
                          /* vector of lengths
} lapi_vec_t;
FORTRAN Syntax
```

```
include 'lapif.h'
LAPI_GETV(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI ADDR TYPE) :: tgt vec
TYPE (LAPI_VEC_T) :: org_vec
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt cntr
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror
The 32-bit version of the LAPI_VEC_T type is defined as:
TYPE LAPI VEC T
  SEQUENCE
```

```
INTEGER(KIND = 4) :: vec type
  INTEGER(KIND = 4) :: num vecs
  INTEGER(KIND = 4) :: info
  INTEGER(KIND = 4) :: len
END TYPE LAPI VEC T
```

The 64-bit version of the LAPI_VEC_T type is defined as:

```
TYPE LAPI VEC T
  SEQUENCE
   INTEGER(KIND = 4) :: vec type
```

```
INTEGER(KIND = 4) :: num_vecs
INTEGER(KIND = 8) :: info
    INTEGER(KIND = 8) :: len
END TYPE LAPI_VEC_T
```

Description

Type of call: point-to-point communication (non-blocking)

This subroutine is the vector version of the LAPI_Get call. Use LAPI_Getv to transfer vectors of data from the target task to the origin task. Both the origin and target vector descriptions are located in the address space of the origin task. But, the values specified in the info array of the target vector must be addresses in the address space of the target task.

The calling program *cannot* assume that the origin buffer can be changed or that the contents of the origin buffers on the origin task are ready for use upon function return. After the origin counter (org_cntr) is incremented, the origin buffers can be modified by the origin task. After the target counter (tgt cntr) is incremented, the target buffers can be modified by the target task. If you provide a completion counter (cmpl cntr), it is incremented at the origin after the target counter (tgt cntr) has been incremented at the target. If the values of any of the counters or counter addresses are NULL (in C) or LAPI ADDR NULL (in FORTRAN), the data transfer occurs, but the corresponding counter increments do not occur.

If any of the following requirements are not met, an error condition occurs:

- The vector types org_vec->vec_type and tgt_vec->vec_type must be the same.
- · If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.
- The length of any vector that is pointed to by tgt_vec must be equal to the length of the corresponding vector that is pointed to by *org_vec*.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the origin side, the contents of the origin buffer are undefined after the operation.

See LAPI_Amsendv for details about communication using different LAPI vector types. (LAPI_Getv does not support the **LAPI_GEN_GENERIC** type.)

Parameters

INPUT

hndl Specifies the LAPI handle.

tat Specifies the task ID of the target task. The value of this parameter must be in the range 0

 $<= tgt < NUM_TASKS.$

tgt_vec Points to the target vector description. Points to the origin vector description. org vec

INPUT/OUTPUT

tgt_cntr Specifies the target counter address. The target counter is incremented once the data

buffer on the target can be modified. If the value of this parameter is NULL (in C) or

LAPI ADDR NULL (in FORTRAN), the target counter is not updated.

Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin org_cntr

> counter is incremented after data arrives at the origin. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

OUTPUT

C Examples

```
To get a LAPI GEN IOVECTOR:
```

```
/* retrieve a remote data buffer address for data to transfer, */
/* such as through LAPI Address init
/* task that calls LAPI_Getv sets up both org_vec and tgt_vec
org_vec->num_vecs = NUM_VECS;
org vec->vec type = LAPI GEN IOVECTOR;
org vec->len = (unsigned long *)
malloc(NUM_VECS*sizeof(unsigned long));
org vec->info = (void **) malloc(NUM VECS*sizeof(void *));
/* each org vec->info[i] gets a base address on the origin task */
/* each org vec->len[i] gets the number of bytes to write to
/* org_vec->info[i]
                                                                */
tgt vec->num vecs = NUM VECS;
tgt vec->vec type = LAPI GEN IOVECTOR;
tgt vec->len = (unsigned long *)
malloc(NUM_VECS*sizeof(unsigned long));
tgt vec->info = (void **) malloc(NUM VECS*sizeof(void *));
/* each tgt_vec->info[i] gets a base address on the target task */
/* each tgt_vec->len[i] gets the number of bytes to transfer
                                                                */
/* from vec->info[i]
                                                                 */
/* For LAPI_GEN_IOVECTOR, num_vecs, vec_type, and len must be
                                                                 */
/* the same
LAPI_Getv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr);
/* tgt cntr and org cntr can both be NULL
/* data will be retrieved as follows:
/* org_vec->len[0] bytes will be retrieved from
/* tgt vec->info[0] and written to org vec->info[0] */
/* org_vec->len[1] bytes will be retrieved from
/* tgt vec->info[1] and written to org vec->info[1] */
/* org vec->len[NUM VECS-1] bytes will be retrieved */
/* from tgt vec->info[NUM_VECS-1] and written to
/* org_vec->info[NUM_VECS-1]
```

For examples of other vector types, see LAPI_Amsendv.

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the *hndl* passed in is not valid (not initialized or in

terminated state).

Indicates that the *org_vec*'s extent (stride * *num_vecs*) is greater than the LAPI_ERR_ORG_EXTENT

value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI ERR ORG STRIDE Indicates that the *org_vec* stride is less than block size. LAPI_ERR_ORG_VEC_ADDR

Indicates that some org vec->info[i] is NULL (in C) or LAPI ADDR NULL (in FORTRAN). but the corresponding length (org_vec->len[i]) is not **0**.

LAPI_ERR_ORG_VEC_LEN Indicates that the total sum of all *org_vec->len[i]* (where [i] is in the range

0 <= *i* <= *org_vec*->*num_vecs*) is greater than the value of LAPI constant

LAPI MAX MSG SZ.

LAPI_ERR_ORG_VEC_NULL Indicates that the org_vec is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN).

LAPI_ERR_ORG_VEC_TYPE Indicates that the *org_vec->vec_type* is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL

Indicates that the strided vector base address org_vec->info[0] is NULL (in

C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL

Indicates that the strided vector address tat vec->info[0] is NULL (in C) or

LAPI ADDR NULL (in FORTRAN).

LAPI ERR TGT Indicates that the tat passed in is outside the range of tasks defined in the

job.

LAPI ERR TGT EXTENT Indicates that tgt vec's extent (stride * num vecs) is greater than the

value of LAPI constant LAPI MAX MSG SZ.

LAPI ERR TGT PURGED Indicates that the subroutine returned early because LAPI Purge totask()

was called.

LAPI ERR TGT STRIDE Indicates that the tgt vec's stride is less than its block size.

LAPI ERR TGT VEC ADDR Indicates that the tgt vec->info[i] is NULL (in C) or LAPI ADDR NULL (in

FORTRAN), but its length (tgt vec->len[i]) is not **0**.

LAPI_ERR_TGT_VEC_LEN Indicates that the sum of tgt_vec->len is greater than the value of LAPI

constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_VEC_NULL Indicates that tgt_vec is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN).

LAPI_ERR_TGT_VEC_TYPE Indicates that the *tgt_vec->vec_type* is not valid.

Indicates that *org_vec* and *tgt_vec* have different lengths (*len*[]). LAPI_ERR_VEC_LEN_DIFF

LAPI_ERR_VEC_NUM_DIFF Indicates that *org_vec* and *tgt_vec* have different *num_vecs*.

LAPI_ERR_VEC_TYPE_DIFF

Indicates that *org_vec* and *tgt_vec* have different vector types (*vec_type*).

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI_Amsendv, LAPI_Getcntr, LAPI_Putv, LAPI_Qenv, LAPI_Waitcntr

LAPI Gfence Subroutine

Purpose

Enforces order on LAPI calls across all tasks and provides barrier synchronization among them.

Library

Availability Library (liblapi_r.a)

C Syntax

#include <lapi.h>

int LAPI_Gfence(hndl) lapi_handle_t hndl;

FORTRAN Syntax

include 'lapif.h'

LAPI GFENCE (hndl, ierror) INTEGER hndl **INTEGER** ierror

Description

Type of call: collective data synchronization (blocking)

Use this subroutine to enforce global order on LAPI calls. This is a collective call. Collective calls must be made in the same order at all participating tasks.

On completion of this call, it is assumed that all LAPI communication associated with hndl from all tasks has guiesced. Although hndl is local, it represents a set of tasks that were associated with it at LAPI_Init, all of which must participate in this operation for it to complete. This is a data fence, which means that the data movement is complete. This is not an operation fence, which would need to include active message completion handlers completing on the target.

Parameters

INPUT

Specifies the LAPI handle. hndl

OUTPUT

Specifies a FORTRAN return code. This is always the last parameter. ierror

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI_Fence

LAPI Init Subroutine

Purpose

Initializes a LAPI context.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Init(hndl,lapi_info)
lapi_handle_t *hndl;
lapi_info_t *lapi_info;
```

FORTRAN Syntax

```
include 'lapif.h'
LAPI_INIT(hndl,lapi_info,ierror)
INTEGER hndl
TYPE (LAPI_INFO_T) :: lapi_info
INTEGER ierror
```

Description

Type of call: Local initialization

Use this subroutine to instantiate and initialize a new LAPI context. A handle to the newly-created LAPI context is returned in hndl. All subsequent LAPI calls can use hndl to specify the context of the LAPI operation. Except for LAPI Address() and LAPI Msg string(), the user cannot make any LAPI calls before calling LAPI_Init().

The lapi info structure (lapi info t) must be "zeroed out" before any fields are filled in. To do this in C, use this statement: bzero (lapi info, size of (lapi info t)). In FORTRAN, you need to "zero out" each field manually in the LAPI INFO T type. Fields with a description of Future support should not be used because the names of those fields might change.

The **lapi_info_t** structure is defined as follows:

```
typedef struct {
                   protocol;
    lapi_dev_t
                                  /* Protocol device returned
                                                                                 */
    lapi_lib_t
                  lib vers:
                                 /* LAPI library version -- user-supplied
    uint
                                 /* No longer used
                   epoch num;
                  num compl hndlr thr; /* Number of completion handler threads
    int
                  instance no; /* Instance of LAPI to initialize [1-16]
                                                                                 */
    uint
                  info6;
                                 /* Future support
                                                                                 */
    int
    LAPI err hndlr *err hndlr:
                                /* User-registered error handler
    com thread info t *lapi thread attr; /* Support thread att and init function */
    void
                  *adapter name; /* What adapter to initialize, i.e. css0, ml0 */
    lapi_extend_t *add info;
                                  /* Additional structure extension
} lapi_info_t;
```

The fields are used as follows:

protocol LAPI sets this field to the protocol that has been initialized.

lib_vers Is used to indicate a library version to LAPI for compatibility purposes. Valid values for this field are:

> L1 LIB Provides basic functionality (this is the default). L2 LIB Provides the ability to use counters as structures.

LAST_LIB Provides the most current level of functionality. For new users of LAPI,

lib_vers should be set to LAST_LIB.

This field must be set to L2_LIB or LAST_LIB to use LAPI_Nopoll_wait and LAPI Setcntr wstatus.

This field is no longer used. epoch_num

num_compl_hndlr_thr

Indicates to LAPI the number of completion handler threads to initialize.

Specifies the instance of LAPI to initialize (1 to 16) instance_no

info6 This field is for future use.

err hndlr Use this field to optionally pass a callback pointer to an error-handler routine.

lapi_thread_attr

Supports thread attributes and initialization function.

adapter_name Is used in persistent subsystem (PSS) mode to pass an adapter name.

Is used for additional information in standalone UDP mode. add info

Parameters

INPUT/OUTPUT

Specifies a structure that provides the parallel job information with which this LAPI context lapi_info

is associated. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL

(in FORTRAN).

OUTPUT

hndl Specifies a pointer to the LAPI handle to initialize.

Specifies a FORTRAN return code. This is always the last parameter. ierror

C Examples

The following environment variable must be set prior to initialization:

```
MP_MSG_API=[ lapi | [ lapi,mpi | mpi,lapi ] | mpi_lapi ]
```

The following environment variables are also commonly used:

```
MP PROCS=number of tasks in job
MP EUILIB=[ ip | us ] (ip is the default)
LAPI USE SHM=[ yes | no | only ] (no is the default)
```

Initialization normally proceeds as follows:

- 1. Set environment variables (done before using application is invoked). Remaining steps are done in user application:
- 2. Clear lapi_info_t, then set any fields.
- 3. Call LAPI Init.

For systems running PE

Both US and IP are supported for shared handles as long as they are the same for both handles. Mixed transport protocols such as LAPI IP and shared user space (US) are not supported.

To initialize a LAPI handle:

```
{
    lapi handle t hndl;
    lapi_info_t info;
   bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */
    LAPI Init(&hndl, &info);
}
To initialize a LAPI handle and register an error handler:
void my_err_hndlr(lapi_handle_t *hndl, int *error_code, lapi_err_t *err_type,
                  int *task id, int *src )
   /* examine passed parameters and delete desired information */
   if ( user wants to terminate ) {
                                        /* will terminate LAPI */
       LAPI Term();
       exit(some return code);
   /* any additional processing */
   return; /* signals to LAPI that error is non-fatal; execution should continue */
{
    lapi handle t hndl;
    lapi info t info;
    bzero(&info, sizeof(lapi info t)); /* clear lapi info */
    /* set error handler pointer */
    info.err hndlr = (LAPI err hndlr) my err hndlr;
    LAPI_Init(&hndl, &info);
}
```

For standalone systems (not running PE)

To initialize a LAPI handle for IP communication using a user handler:

```
int my_udp_hndlr(lapi_handle_t *hndl, lapi_udp_t *local_addr, lapi_udp_t *addr_list,
                 lapi udpinfo t *info)
{
    /* LAPI will allocate and free addr_list pointer when using */
   /* a user handler
    /* use the AIX inet addr call to convert an IP address
    /* from a dotted quad to a long
    task_0_ip_as_long = inet_addr(task_0_ip_as_string);
    addr list[0].ip addr = task 0 ip as long;
    addr list[0].port no = task 0 port as unsigned;
    task 1 ip as long = inet addr(task 1 ip as string);
    addr_list[1].ip_addr = task_1_ip_as_long;
    addr_list[1].port_no = task_1_port_as_unsigned;
    task num tasks-1 ip as long = inet addr(task num tasks-1 ip as string);
    addr_list[num_tasks-1].ip_addr = task_num_tasks-1_ip_as_long;
```

```
addr list[num tasks-1].port no = task num tasks-1 port as unsigned;
}
   lapi handle t hndl;
   lapi info_t info;
   lapi_extend_t extend_info;
   extend_info.udp_hndlr = (udp_init_hndlr *) my_udp_hndlr;
   info.add info = &extend info;
   LAPI Init(&hndl, &info);
To initialize a LAPI handle for IP communication using a user list:
    lapi handle t hndl;
    lapi info t info;
    lapi_extend_t extend_info;
   lapi_udp_t *addr_list;
   bzero(&info, sizeof(lapi info t));
                                              /* clear lapi info
   bzero(&extend_info, sizeof(lapi_extend_t)); /* clear lapi_extend_info */
    /* when using a user list, the user is responsible for allocating
    /* and freeing the list pointer
   addr list = malloc(num tasks);
    /* Note, since we need to know the number of tasks before LAPI is
   /* initialized, we can't use LAPI Qenv. getenv("MP PROCS") will
   /* do the trick.
   /* populate addr_list
    /* use the AIX inet addr call to convert an IP address
   /* from a dotted quad to a long
    task_0_ip_as_long = inet_addr(task_0_ip_as_string);
   addr_list[0].ip_addr = task_0_ip_as_long;
   addr_list[0].port_no = task_0_port_as_unsigned;
    task_1_ip_as_long = inet_addr(task_1_ip_as_string);
    addr_list[1].ip_addr = task_1_ip_as_long;
   addr_list[1].port_no = task_1_port_as_unsigned;
    task num tasks-1 ip as long = inet addr(task num tasks-1 ip as string);
    addr list[num tasks-1].ip addr = task num tasks-1 ip as long;
    addr list[num tasks-1].port no = task num tasks-1 port as unsigned;
    /* then assign to extend pointer */
    extend info.add udp addrs = addr list;
    info.add_info = &extend_info;
    LAPI_Init(&hndl, &info);
```

```
/* user's responsibility only in the case of user list */
    free(addr list);
}
```

See the LAPI sample programs for complete examples of initialization in standalone mode.

To initialize a LAPI handle for user space communication in standalone mode:

```
export MP MSG API=lapi
export MP EUILIB=us
export MP PROCS=
                                   /* number of tasks in job
export MP PARTITION=
                                   /* unique job key
export MP CHILD=
                                   /* unique task ID
export MP_LAPI NETWORK=@1:164,sn0 /* LAPI network information */
run LAPI jobs as normal
```

See the README.LAPI.STANDALONE.US file in the standalone/us directory of the LAPI sample files for complete details.

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_ALL_HNDL_IN_USE

All available LAPI instances are in use.

LAPI ERR BOTH NETSTR SET

Both the MP_LAPI_NETWORK and MP_LAPI_INET statements are set (only one should be set).

LAPI_ERR_CSS_LOAD_FAILED

LAPI is unable to load the communication utility library.

LAPI ERR HNDL INVALID

The lapi handle t * passed to LAPI for initialization is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI ERR INFO NONZERO INFO

The future support fields in the lapi_info_t structure that was passed to LAPI are not set to zero (and should be).

LAPI ERR INFO NULL

The **lapi info t** pointer passed to LAPI is NULL (in C) or LAPI ADDR NULL (in FORTRAN).

LAPI_ERR_MEMORY_EXHAUSTED

LAPI is unable to obtain memory from the system.

LAPI ERR MSG API

Indicates that the MP MSG API environment variable is not set correctly.

LAPI ERR NO NETSTR SET

No network statement is set. Note that if running with POE, this will be returned if MP_MSG_API is not set correctly.

LAPI_ERR_NO_UDP_HNDLR You passed a value of NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) for both the UDP handler and the UDP list. One of these (the UDP handler or the UDP list) must be initialized for standalone UDP initialization. This error is returned in standalone UDP mode only.

LAPI_ERR_PSS_NON_ROOT You tried to initialize the persistent subsystem (PSS) protocol as a non-root user.

LAPI_ERR_SHM_KE_NOT_LOADED

LAPI's shared memory kernel extension is not loaded.

LAPI_ERR_SHM_SETUP LAPI is unable to set up shared memory. This error will be returned if

LAPI USE SHM=only and tasks are assigned to more than one node.

The UDP packet size you indicated is not valid. LAPI ERR UDP PKT SZ

LAPI_ERR_UNKNOWN An internal error has occurred.

LAPI_ERR_USER_UDP_HNDLR_FAIL

The UDP handler you passed has returned a non-zero error code. This

error is returned in standalone UDP mode only.

Location

/usr/lib/liblapi r.a

Related Information

RSCT for AIX 5L: LAPI Programming Guide contains information about

- Initializing LAPI
- Bulk message transfer

Subroutines: LAPI_Msg_string, LAPI_Term

LAPI_Msg_string Subroutine

Purpose

Retrieves the message that is associated with a subroutine return code.

Library

Availability Library (liblapi_r.a)

C Syntax

#include <lapi.h>

LAPI Msg string(error code, buf) int error code; void *buf;

FORTRAN Syntax

include 'lapif.h'

LAPI MSG STRING(error code, buf, ierror) INTEGER error code CHARACTER buf (LAPI_MAX_ERR_STRING) **INTEGER** ierror

Description

Type of call: local queries

Use this subroutine to retrieve the message string that is associated with a LAPI return code. LAPI tries to find the messages of any return codes that come from the AIX operating system or its communication subsystem.

Parameters

INPUT

error_code Specifies the return value of a previous LAPI call.

OUTPUT

buf Specifies the buffer to store the message string.

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

```
To get the message string associated with a LAPI return code:
```

```
char msg buf[LAPI MAX ERR STRING]; /* constant defined in lapi.h */
    int rc, errc;
    rc = some LAPI call();
    errc = LAPI Msg string(rc, msg buf);
    /* msg buf now contains the message string for the return code */
}
```

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI ERR CATALOG FAIL Indicates that the message catalog cannot be opened. An English-only

string is copied into the user's message buffer (buf).

LAPI_ERR_CODE_UNKNOWN

Indicates that error_code is outside of the range known to LAPI.

LAPI_ERR_RET_PTR_NULL Indicates that the value of the buf pointer is NULL (in C) or that the value

of buf is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi r.a

Related Information

RSCT for AIX 5L: LAPI Programming Guide contains information about

- Initializing LAPI
- · Bulk message transfer

Subroutines: LAPI_Msg_string, LAPI_Term

LAPI_Msgpoll Subroutine

Purpose

Allows the calling thread to check communication progress.

Library

Availability Library (liblapi_r.a)

C Syntax

FORTRAN Syntax

```
include 'lapif.h'
LAPI_MSGPOLL(hndl, cnt, info, ierror)
INTEGER hndl
INTEGER cnt
TYPE (LAPI_MSG_STATE_T) :: info
INTEGER ierror
```

Description

Type of call: local progress monitor (blocking)

The **LAPI_MsgpoII** subroutine allows the calling thread to check communication progress. With this subroutine, LAPI provides a means of running the dispatcher several times until either progress is made or a specified maximum number of dispatcher loops have executed. Here, *progress* is defined as the completion of either a message send operation or a message receive operation.

LAPI_MsgpoII is intended to be used when interrupts are turned off. If the user has not explicitly turned interrupts off, LAPI temporarily disables interrupt mode while in this subroutine because the dispatcher is called, which will process any pending receive operations. If the LAPI dispatcher loops for the specified maximum number of times, the call returns. If progress is made before the maximum count, the call will return immediately. In either case, LAPI will report status through a data structure that is passed by reference.

The **lapi_msg_info_t** structure contains a flags field (*status*), which is of type **lapi_msg_state_t**. Flags in the *status* field are set as follows:

LAPI DISP CNTR If the dispatcher has looped *cnt* times without making progress

LAPI_SEND_COMPLETE If a message send operation has completed

LAPI RECV COMPLETE If a message receive operation has completed

LAPI_BOTH_COMPLETE If both a message send operation and a message receive operation have

completed

LAPI_POLLING_NET If another thread is already polling the network or shared memory

completion

Parameters

INPUT

hndl Specifies the LAPI handle.

cnt Specifies the maximum number of times the dispatcher should loop with no progress

before returning.

info Specifies a status structure that contains the result of the LAPI_Msgpoll() call.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To loop through the dispatcher no more than 1000 times, then check what progress has been made:

Return Values

}

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_MSG_INFO_NULL

Indicates that the *info* pointer is NULL (in C) or that the value of *info* is **LAPI ADDR NULL** (in FORTRAN).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Getcntr, LAPI_Probe, LAPI_Setcntr, LAPI_Waitcntr

LAPI_Nopoll_wait Subroutine

Purpose

Waits for a counter update without polling.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
void LAPI_Nopoll_wait(hndl, cntr_ptr, val, cur_cntr_val)
lapi_handle_t hndl;
lapi_cntr_t *cntr_ptr;
int val;
int *cur_cntr_val;
```

FORTRAN Syntax

```
include 'lapif.h'
int LAPI_NOPOLL_WAIT(hndl, cntr, val, cur_cntr_val, ierror)
INTEGER hndl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER val
INTEGER cur_cntr_val
INTEGER ierror
```

Description

Type of call: recovery

This subroutine waits for a counter update without polling (that is, without explicitly invoking LAPI's internal communication dispatcher). This call may or may not check for message arrivals over the LAPI context hndl. The cur_cntr_val variable is set to the current counter value. Although it has higher latency than LAPI_Waitcntr, LAPI_Nopoll_wait frees up the processor for other uses.

Note: To use this subroutine, the *lib_vers* field in the **lapi_info_t** structure must be set to **L2_LIB** or **LAST_LIB**.

Parameters

INPUT

hndl Specifies the LAPI handle.

val Specifies the relative counter value (starting from 1) that the counter needs to reach

before returning.

cur cntr val Specifies the integer value of the current counter. The value of The value of this parameter

can be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

INPUT/OUTPUT

cntr_ptrPoints to the lapi_cntr_t structure in C.cntrls the lapi_cntr_t structure in FORTRAN.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

Use of this subroutine is not recommended on a system that is running Parallel Environment (PE).

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL Indicates that the *cntr_ptr* pointer is NULL (in C) or that the value of *cntr* is

LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_CNTR_VAL Indicates that the *val* passed in is less than or equal to **0**.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_MULTIPLE_WAITERS

Indicates that more than one thread is waiting for the counter.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask()

was called.

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI_Init, LAPI_Purge_totask, LAPI_Resume_totask, LAPI_Setcntr_wstatus

LAPI_Probe Subroutine

Purpose

Transfers control to the communication subsystem to check for arriving messages and to make progress in polling mode.

Library

Availability Library (liblapi_r.a)

C Syntax

#include <lapi.h>

int LAPI_Probe(hndl)
lapi_handle_t hndl;

FORTRAN Syntax

include 'lapif.h'

int LAPI_PROBE(hndl, ierror)
INTEGER hndl
INTEGER ierror

Description

Type of call: local progress monitor (non-blocking)

This subroutine transfers control to the communication subsystem in order to make progress on messages associated with the context *hndl*. A **LAPI_Probe** operation lasts for one round of the communication dispatcher.

Note: There is no guarantee about receipt of messages on the return from this function.

Parameters

INPUT

hndl Specifies the LAPI handle.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI ERR HNDL INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI Getcntr, LAPI Msgpoll, LAPI Nopoll wait, LAPI Waitcntr

LAPI_Purge_totask Subroutine

Purpose

Allows a task to cancel messages to a given destination.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Purge_totask(hndl, dest)
lapi_handle_t hndl;
uint
              dest;
```

FORTRAN Syntax

```
include 'lapif.h'
int LAPI PURGE TOTASK(hndl, dest, ierror)
INTEGER hndl
INTEGER dest
INTEGER ierror
```

Description

Type of call: recovery

This subroutine cancels messages and resets the state corresponding to messages in flight or submitted to be sent to a particular target task. This is an entirely local operation. For correct behavior a similar invocation is expected on the destination (if it exists). This function cleans up all the state associated with pending messages to the indicated target task. It is assumed that before the indicated task starts communicating with this task again, it also purges this instance (or that it was terminated and initialized again). It will also wake up all threads that are in LAPI_Nopoll_wait depending on how the arguments are passed to the LAPI_Nopoll_wait function. The behavior of LAPI_Purge_totask is undefined if LAPI collective functions are used.

Note: This subroutine should not be used when the parallel application is running in a PE/LoadLeveler environment.

LAPI_Purge_totask is normally used after connectivity has been lost between two tasks. If connectivity is restored, the tasks can restored for LAPI communication by calling **LAPI_Resume_totask**.

Parameters

INPUT

hndl Specifies the LAPI handle.

dest Specifies the destination instance ID to which pending messages need to be cancelled.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

Use of this subroutine is not recommended on a system that is running Parallel Environment (PE).

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_TGT Indicates that *dest* is outside the range of tasks defined in the job.

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Init, LAPI_Nopoll_wait, LAPI_Resume_totask, LAPI_Term

LAPI_Put Subroutine

Purpose

Transfers data from a local task to a remote task.

Library

Availability Library (liblapi_r.a)

C Syntax

FORTRAN Syntax

```
include 'lapif.h'
int LAPI PUT(hndl, tgt, len, tgt addr, org addr, tgt cntr, org cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_LONG TYPE) :: len
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_addr
INTEGER org addr
INTEGER (KIND=LAPI ADDR TYPE) :: tgt cntr
TYPE (LAPI_CNTR_T) :: org cntr
TYPE (LAPI_CNTR_T) :: cmpl cntr
INTEGER ierror
```

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to transfer data from a local (origin) task to a remote (target) task. The origin counter will increment on the origin task upon origin buffer availability. The target counter will increment on the target and the completion counter will increment at the origin task upon message completion. Because there is no completion handler, message completion and target buffer availability are the same in this case.

This is a non-blocking call. The caller *cannot* assume that the data transfer has completed upon the return of the function. Instead, counters should be used to ensure correct buffer accesses as defined above.

Note that a zero-byte message does not transfer data, but it does have the same semantic with respect to counters as that of any other message.

Parameters

INPUT

hndl	Specifies the LAPI handle.
tgt	Specifies the task ID of the target task. The value of this parameter must be in the range $c = tgt < NUM_TASKS$.
len	Specifies the number of bytes to be transferred. This parameter must be in the range 0 <= len <= the value of LAPI constant LAPI_MAX_MSG_SZ .
tgt_addr	Specifies the address on the target task where data is to be copied into. If <i>len</i> is 0 , The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
org_addr	Specifies the address on the origin task from which data is to be copied. If <i>len</i> is 0 , The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
INIDIIT/OUTDI	_

org_addr	value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).		
INPUT/OUTPUT			
tgt_cntr	Specifies the target counter address. The target counter is incremented upon message completion. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.		
org_cntr	Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented at buffer availability. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.		
cmpl_cntr	Specifies the completion counter address (in C) or the completion counter (in FORTRAN) that is a reflection of <i>tgt_cntr</i> . The completion counter is incremented at the origin after <i>tgt_cntr</i> is incremented. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the completion counter is not updated.		

OUTPUT

ierror

Specifies a FORTRAN return code. This is always the last parameter.

C Examples

```
{
    /* initialize the table buffer for the data addresses
                                                                   */
    /* get remote data buffer addresses
    LAPI_Address_init(hndl,(void *)data_buffer,data_buffer_list);
    LAPI Put(hndl, tgt, (ulong) data len, (void *)(data buffer list[tgt]),
             (void *) data buffer, tgt cntr, org cntr, compl cntr);
    /* transfer data len bytes from local address data buffer.
    /* write the data starting at address data buffer list[tgt] on */
    /* task tgt. tgt cntr, org cntr, and compl cntr can be NULL. */
```

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_DATA_LEN Indicates that the value of len is greater than the value of LAPI constant

LAPI_MAX_MSG_SZ.

LAPI ERR HNDL INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI ERR ORG ADDR NULL

Indicates that the org_addr parameter passed in is NULL (in C) or **LAPI ADDR NULL** (in FORTRAN), but *len* is greater than **0**.

LAPI ERR TGT Indicates that the tat passed in is outside the range of tasks defined in the

job.

LAPI_ERR_TGT_ADDR_NULL

Indicates that the tgt_addr parameter passed in is NULL (in C) or **LAPI ADDR NULL** (in FORTRAN), but *len* is greater than **0**.

LAPI ERR TGT PURGED Indicates that the subroutine returned early because LAPI Purge totask()

was called.

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI Get, LAPI Getcntr, LAPI Qenv, LAPI Setcntr, LAPI Waitcntr, LAPI Xfer

LAPI_Putv Subroutine

Purpose

Transfers vectors of data from a local task to a remote task.

Library

Availability Library (liblapi_r.a)

C Syntax

#include <lapi.h>

```
int LAPI_Putv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, cmpl_cntr)
lapi_handle_t hndl;
uint
              tgt;
            *tgt_vec;
lapi vec t
lapi vec t
             *org vec;
lapi_cntr_t *tgt_cntr;
lapi_cntr_t *org cntr;
lapi_cntr_t *cmpl cntr;
typedef struct {
   lapi_vectype_t vec_type; /* operation code
   uint num_vecs; /* number of vectors */
void **info; /* vector of information */
ulong *len; /* vector of lengths */
   ulong
} lapi_vec_t;
FORTRAN Syntax
include 'lapif.h'
LAPI_PUTV(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, cmpl_cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt vec
TYPE (LAPI_VEC_T) :: org vec
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl cntr
INTEGER ierror
The 32-bit version of the LAPI VEC T type is defined as:
TYPE LAPI VEC T
  SEQUENCE
  INTEGER(KIND = 4) :: vec type
  INTEGER(KIND = 4) :: num_vecs
  INTEGER(KIND = 4) :: info
  INTEGER(KIND = 4) :: len
END TYPE LAPI VEC T
The 64-bit version of the LAPI_VEC_T type is defined as:
TYPE LAPI VEC T
  SEQUENCE
  INTEGER(KIND = 4) :: vec type
  INTEGER(KIND = 4) :: num vecs
  INTEGER(KIND = 8) :: info
  INTEGER(KIND = 8) :: len
END TYPE LAPI VEC T
```

Description

Type of call: point-to-point communication (non-blocking)

LAPI_Putv is the vector version of the **LAPI_Put** call. Use this subroutine to transfer vectors of data from the origin task to the target task. The origin vector descriptions and the target vector descriptions are

located in the address space of the *origin* task. However, the values specified in the *info* array of the target vector must be addresses in the address space of the target task.

The calling program cannot assume that the origin buffer can be changed or that the contents of the target buffers on the target task are ready for use upon function return. After the origin counter (org. cntr) is incremented, the origin buffers can be modified by the origin task. After the target counter (tat cntr) is incremented, the target buffers can be modified by the target task. If you provide a completion counter (cmpl cntr), it is incremented at the origin after the target counter (tgt cntr) has been incremented at the target. If the values of any of the counters or counter addresses are NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the data transfer occurs, but the corresponding counter increments do not occur.

If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.

The length of any vector pointed to by org_vec must be equal to the length of the corresponding vector pointed to by tgt_vec.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the target side, the contents of the target buffer are undefined after the operation.

See LAPI Amsendy for more information about using the various vector types. (LAPI Puty does not support the LAPI GEN GENERIC type.)

Parameters

INPUT

hndl Specifies the LAPI handle.

tat Specifies the task ID of the target task. The value of this parameter must be in the range 0

<= tgt < NUM TASKS.

Points to the target vector description. tgt_vec Points to the origin vector description. org_vec

INPUT/OUTPUT

Specifies the target counter address. The target counter is incremented upon message tgt cntr

completion. If this parameter is NULL (in C) or LAPI ADDR NULL (in FORTRAN), the

target counter is not updated.

Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin org_cntr

counter is incremented at buffer availability. If this parameter is NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

cmpl_cntr Specifies the completion counter address (in C) or the completion counter (in FORTRAN)

> that is a reflection of tgt_cntr. The completion counter is incremented at the origin after tgt_cntr is incremented. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN), the completion counter is not updated.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

C Examples

```
To put a LAPI_GEN_IOVECTOR:
     /* retrieve a remote data buffer address for data to transfer, */
```

```
/* such as through LAPI Address init
                                                               */
/* task that calls LAPI Putv sets up both org vec and tgt vec
org_vec->num_vecs = NUM_VECS;
org_vec->vec_type = LAPI_GEN_IOVECTOR;
             = (unsigned long *)
org vec->len
malloc(NUM VECS*sizeof(unsigned long));
org vec->info = (void **) malloc(NUM VECS*sizeof(void *));
/* each org_vec->info[i] gets a base address on the origin task */
/* each org vec->len[i] gets the number of bytes to transfer
/* from org vec->info[i]
tgt vec->num vecs = NUM VECS;
tgt_vec->vec_type = LAPI GEN IOVECTOR;
tgt vec->len
             = (unsigned long *)
malloc(NUM VECS*sizeof(unsigned long));
tgt vec->info = (void **) malloc(NUM VECS*sizeof(void *));
/* each tgt_vec->info[i] gets a base address on the target task
/* each tgt vec->len[i] gets the number of bytes to write to vec->info[i] */
/* For LAPI GEN IOVECTOR, num vecs, vec type, and len must be the same
LAPI Putv(hndl, tgt, tgt vec, org vec, tgt cntr, org cntr, compl cntr);
/* tgt cntr, org cntr and compl cntr can all be NULL */
/* data will be transferred as follows:
/* org vec->len[0] bytes will be retrieved from
                                                    */
/* org_vec->info[0] and written to tgt vec->info[0]
                                                    */
/* org_vec->len[1] bytes will be retrieved from
/* org vec->info[1] and written to tgt vec->info[1]
/* org vec->len[NUM VECS-1] bytes will be retrieved */
/* from org_vec->info[NUM_VECS-1] and written to
/* tgt_vec->info[NUM_VECS-1]
```

See the example in **LAPI_Amsendv** for information on other vector types.

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_ORG_EXTENT Indicates that the *org_vec*'s extent (stride * *num_vecs*) is greater than the

value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_STRIDE Indicates that the *org_vec* stride is less than block.

LAPI_ERR_ORG_VEC_ADDR

Indicates that the *org_vec->info[i]* is NULL (in C) or **LAPI_ADDR_NULL** (in

FORTRAN), but its length (org_vec->len[i]) is not **0**.

LAPI_ERR_ORG_VEC_LEN Indicates that the sum of org_vec->len is greater than the value of LAPI

constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_VEC_NULL Indicates that the org_vec is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN).

LAPI_ERR_ORG_VEC_TYPE Indicates that the *org_vec->vec_type* is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL

Indicates that the strided vector address $org_vec ext{->}info[0]$ is NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL

Indicates that the strided vector address tgt_vec->info[0] is NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT Indicates that the *tgt* passed in is outside the range of tasks defined in the

job.

LAPI_ERR_TGT_EXTENT Indicates that *tgt_vec*'s extent (stride * *num_vecs*) is greater than the

value of LAPI constant LAPI MAX MSG SZ.

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask()

was called.

LAPI_ERR_TGT_STRIDE Indicates that the *tgt_vec* stride is less than block.

LAPI_ERR_TGT_VEC_ADDR Indicates that the tgt_vec->info[i] is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN), but its length (*tgt_vec->len*[i]) is not **0**.

LAPI_ERR_TGT_VEC_LEN Indicates that the sum of *tgt_vec->len* is greater than the value of LAPI

constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_VEC_NULL Indicates that tgt_vec is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN).

LAPI_ERR_TGT_VEC_TYPE Indicates that the *tgt_vec->vec_type* is not valid.

LAPI_ERR_VEC_LEN_DIFF Indicates that *org_vec* and *tgt_vec* have different lengths (*len[]*).

LAPI_ERR_VEC_NUM_DIFF Indicates that *org_vec* and *tgt_vec* have different *num_vecs*.

LAPI_ERR_VEC_TYPE_DIFF

Indicates that *org_vec* and *tgt_vec* have different vector types (*vec_type*).

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI_Amsendv, LAPI_Getcntr, LAPI_Getv, LAPI_Qenv, LAPI_Setcntr, LAPI_Waitcntr, LAPI Xfer

LAPI_Qenv Subroutine

Purpose

Used to query LAPI for runtime task information.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapif.h>
int LAPI_Qenv(hndl, query, ret_val)
lapi_handle_t hndl;
lapi_query_t query;
int *ret_val; /* ret_val's type varies (see Additional query types) */
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_QENV(hndl, query, ret_val, ierror)
INTEGER hndl
INTEGER query
INTEGER ret_val /* ret_val's type varies (see Additional query types) */
INTEGER ierror
```

Description

Type of call: local queries

Use this subroutine to query runtime settings and statistics from LAPI. LAPI defines a set of query types as an enumeration in **lapi.h** for C and explicitly in the 32-bit and 64-bit versions of **lapif.h** for FORTRAN.

For example, you can query the size of the table that LAPI uses for the **LAPI_Addr_set** subroutine using a *query* value of **LOC_ADDRTBL_SZ**:

```
LAPI_Qenv(hndl, LOC_ADDRTBL_SZ, &ret val);
```

ret_val will contain the upper bound on the table index. A subsequent call to LAPI_Addr_set (hndl, addr, addr_hndl); could then ensure that the value of addr_hndl is between 0 and ret_val.

When used to show the size of a parameter, a comparison of values, or a range of values, valid values for the *query* parameter of the **LAPI_Qenv** subroutine appear in <u>underlined</u>, <u>bold uppercase</u> characters, for example:

LOC_ADDRTBL_SZ

This typographic convention is a shorthand notation for:

```
LAPI_Qenv(hndl, LOC_ADDRTBL_SZ, ret_val)
```

In C, **lapi_query_t** defines the valid types of LAPI queries:

```
typedef enum {
        TASK ID=0.
                        /* Query the task ID of the current task in the job
                                                                                     */
        NUM TASKS,
                       /* Query the number of tasks in the job
        MAX UHDR SZ,
                      /* Query the maximum user header size for active messaging
        MAX DATA SZ,
                       /* Query the maximum data length that can be sent
                                                                                     */
        ERROR CHK,
                        /* Query and set parameter checking on (1) or off (0)
                                                                                     */
        TIMEOUT,
                        /* Query and set the current communication timeout setting */
                        /* in seconds
        MIN TIMEOUT, /* Query the minimum communication timeout setting in seconds */
        MAX TIMEOUT, /* Query the maximum communication timeout setting in seconds */
        INTERRUPT_SET, /* Query and set interrupt mode on (1) or off (0)
        MAX PORTS, /* Query the maximum number of available communication ports */
        MAX PKT SZ,
                       /* This is the payload size of 1 packet
                                                                                     */
        NUM_REX_BUFS, /* Number of retransmission buffers
                                                                                     */
                      /* Size of each retransmission buffer in bytes
        REX BUF SZ,
                                                                                     */
        LOC ADDRTBL SZ, /* Size of address store table used by LAPI Addr set
                                                                                     */
                      /* No longer used by LAPI (supports legacy code)
/* No longer used by LAPI (supports legacy code)
        EPOCH NUM,
        USE THRESH,
                                                                                     */
        RCV FIFO SIZE, /* No longer used by LAPI (supports legacy code)
        MAX ATOM SIZE,/* Query the maximum atom size for a DGSP accumulate transfer*/
        BUF CP SIZE, /* Query the size of the message buffer to save (default 128b)*/
        MAX PKTS OUT, /* Query the maximum number of messages outstanding /
                                                                                     */
                        /* destination
                                                                                     */
        ACK THRESHOLD, /* Query and set the threshold of acknowledgments going
                                                                                     */
                        /* back to the source
                                                                                     */
        QUERY SHM ENABLED,
                             /* Query to see if shared memory is enabled
```

```
QUERY SHM NUM TASKS, /* Query to get the number of tasks that use shared */
          QUERY SHM TASKS, /* Query to get the list of task IDs that make up shared */
                                /* memory; pass in an array of size QUERY_SHM_NUM_TASKS */
          QUERY STATISTICS,
                                    /* Query to get packet statistics from LAPI, as
                                                                                                       */
                                    /* defined by the lapi_statistics_t structure. For
                                    /* this query, pass in 'lapi statistics t *' rather
                                    /* than 'int *ret_val'; otherwise, the data will
                                                                                                       */
                                   /* overflow the buffer.
                                                                                                       */
          PRINT STATISTICS.
                                   /* Query debug print function to print out statistics */
          VERYLARGE MSG THRESH, /* For future use
          LAST QUERY
} lapi query t;
typedef struct {
        lapi long t Tot dup pkt cnt;
                                             /* Total duplicate packet count */
        lapi_long_t Tot_retrans_pkt_cnt; /* Total retransmit packet count */
        lapi_long_t Tot_gho_pkt_cnt; /* Total Ghost packet count
lapi_long_t Tot_pkt_sent_cnt; /* Total packet sent count
lapi_long_t Tot_pkt_recv_cnt; /* Total packet receive count
lapi_long_t Tot_data_sent; /* Total data sent
lapi_long_t Tot_data_recv; /* Total data receive
                                                                                          */
                                                                                           */
                                                                                           */
                                                                                           */
       } lapi statistics t;
```

In FORTRAN, the valid types of LAPI gueries are defined in **lapif.h** as follows:

```
INTEGER TASK ID, NUM TASKS, MAX UHDR SZ, MAX DATA SZ, ERROR CHK
      INTEGER TIMEOUT, MIN TIMEOUT, MAX TIMEOUT
      INTEGER INTERRUPT_SET, MAX_PORTS, MAX_PKT_SZ, NUM_REX_BUFS
      INTEGER REX BUF SZ,LOC ADDRTBL SZ,EPOCH NUM, USE THRESH
      INTEGER RCV FIFO SIZE, MAX ATOM SIZE, BUF CP SIZE
      INTEGER MAX PKTS OUT, ACK THRESHOLD, QUERY SHM ENABLED
      INTEGER QUERY SHM NUM TASKS, QUERY SHM TASKS
      INTEGER QUERY_STATISTICS, PRINT_STATISTICS
      INTEGER VERYLARGE MSG THRESH
      INTEGER LAST QUERY
      PARAMETER (TASK ID=0, NUM TASKS=1, MAX UHDR SZ=2, MAX DATA SZ=3)
      PARAMETER (ERROR CHK=4, TIMEOUT=5, MIN TIMEOUT=6)
      PARAMETER (MAX TIMEOUT=7, INTERRUPT SET=8, MAX PORTS=9)
      PARAMETER (MAX_PKT_SZ=10, NUM_REX_BUFS=11, REX_BUF_SZ=12)
      PARAMETER (LOC ADDRTBL SZ=13, EPOCH NUM=14, USE THRESH=15)
      PARAMETER (RCV FIFO SIZE=16, MAX ATOM SIZE=17, BUF CP SIZE=18)
      PARAMETER (MAX_PKTS_OUT=19,ACK_THRESHOLD=20)
      PARAMETER (QUERY SHM ENABLED=21, QUERY SHM NUM TASKS=22)
      PARAMETER (QUERY SHM TASKS=23, QUERY STATISTICS=24)
      PARAMETER (PRINT STATISTICS=25)
      PARAMETER (VERYLARGE MSG THRESH=26)
      PARAMETER (LAST QUERY=27)
```

Additional query types

LAPI provides additional query types for which the behavior of LAPI_Qenv is slightly different:

QUERY STATISTICS

When passed this query type, LAPI Qenv interprets ret val as a pointer to type lapi statistics t. Upon function return, the fields of the structure contain LAPI's data transfer statistics. Note that the statistics are only with respect to data transferred using the user space (US) or UDP/IP protocols. Data transferred through shared memory or intra-task local copy are not reflected.

PRINT_STATISTICS

When passed this query type, LAPI sends data transfer statistics to standard output. In this case, ret_val is unaffected. However, LAPI's error checking requires that the value of ret_val is not NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) for all LAPI_Qenv types (including PRINT STATISTICS).

QUERY_SHM_TASKS When passed this query type, LAPI_Qenv returns a list of task IDs with

which this task can communicate using shared memory. *ret_val* must be an **int** * with enough space to hold **NUM_TASKS** integers. For each task *i*, if it is possible to use shared memory, *ret_val[i]* will contain the shared memory task ID. If it is not possible to use shared memory, *ret_val[i]* will contain **-1**.

Parameters

INPUT

hndl Specifies the LAPI handle.

query Specifies the type of query you want to request. In C, the values for query are defined by

the lapi_query_t enumeration in lapi.h. In FORTRAN, these values are defined explicitly

in the 32-bit version and the 64-bit version of lapif.h.

OUTPUT

ret_val Specifies the reference parameter for LAPI to store as the result of the query. The value of

this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

```
To query runtime values from LAPI:
```

```
int task_id;
lapi_statistics_t stats;
.
.
.
.
LAPI_Qenv(hndl, TASK_ID, &task_id);
/* task_id now contains the task ID */
.
.
.
.
LAPI_Qenv(hndl, QUERY_STATISTICS, (int *)&stats);
/* the fields of the stats structure are now
    filled in with runtime values */
.
.
.
```

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_QUERY_TYPE Indicates that the query passed in is not valid.

LAPI_ERR_RET_PTR_NULL Indicates that the value of the ret_val pointer is NULL (in C) or that the

value of *ret_val* is **LAPI_ADDR_NULL** (in FORTRAN).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Amsend, LAPI_Get, LAPI_Put, LAPI_Senv, LAPI_Xfer

LAPI Resume totask Subroutine

Purpose

Re-enables the sending of messages to the task.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Resume_totask(hndl, dest)
lapi_handle_t hndl;
uint dest;
```

FORTRAN Syntax

```
include 'lapif.h'
```

```
int LAPI_RESUME_TOTASK(hndl, dest, ierror)
INTEGER hndl
INTEGER dest
INTEGER ierror
```

Description

Type of call: recovery

This subroutine is used in conjunction with **LAPI_Purge_totask**. It enables LAPI communication to be reestablished for a task that had previously been purged. The purged task must either restart LAPI or execute a **LAPI_Purge_totask/LAPI_Resume_totask** sequence for this task.

Parameters

INPUT

hndl Specifies the LAPI handle.

dest Specifies the destination instance ID with which to resume communication.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

Use of this subroutine is not recommmended on a system that is running Parallel Environment (PE).

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_TGT Indicates that the *tgt* passed in is outside the range of tasks defined in the

job.

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Init, LAPI_Nopoll_wait, LAPI_Purge_totask, LAPI_Term

LAPI_Rmw Subroutine

Purpose

Provides data synchronization primitives.

Library

Availability Library (liblapi r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Rmw(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr)
lapi_handle_t hndl;
RMW_ops_t op;
uint tgt;
int *tgt_var;
int *in_val;
int *prev_tgt_val;
lapi_cntr_t *org_cntr;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_RMW(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr, ierror)
INTEGER hndl
INTEGER op
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_var
INTEGER in_val
INTEGER prev_tgt_val
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror
```

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to synchronize two independent pieces of data, such as two tasks sharing a common data structure. The operation is performed at the target task (*tgt*) and is atomic. The operation takes an input value (*in_val*) from the origin and performs one of four operations (*op*) on a variable (*tgt_var*) at the target (*tgt*), and then replaces the target variable (*tgt_var*) with the results of the operation (*op*). The original value (*prev_tgt_val*) of the target variable (*tgt_var*) is returned to the origin.

The operations (*op*) are performed over the context referred to by *hndl*. The outcome of the execution of these calls is as if the following code was executed atomically:

```
*prev_tgt_val = *tgt_var;
*tgt_var = f(*tgt_var, *in_val);
where:

f(a,b) = a + b for FETCH_AND_ADD

f(a,b) = a | b for FETCH_AND_OR (bitwise or)

f(a,b) = b for SWAP
```

For **COMPARE_AND_SWAP**, *in_val* is treated as a pointer to an array of two integers, and the *op* is the following atomic operation:

```
if(*tgt_var == in_val[0]) {
   *prev_tgt_val = TRUE;
   *tgt_var = in_val[1];
} else {
   *prev_tgt_val = FALSE;
}
```

All **LAPI_Rmw** calls are non-blocking. To test for completion, use the **LAPI_Getcntr** and **LAPI_Waitcntr** subroutines. **LAPI_Rmw** does not include a target counter (*tgt_cntr*), so **LAPI_Rmw** calls do not provide any indication of completion on the target task (*tgt*).

Parameters

INPUT

hndl Specifies the LAPI handle.

op Specifies the operation to be performed. The valid operations are:

- COMPARE_AND_SWAP
- FETCH_AND_ADD
- FETCH AND OR
- SWAP

Specifies the task ID of the target task where the read-modify-write (Rmw) variable

resides. The value of this parameter must be in the range $0 \le tgt < NUM_TASKS$.

tgt_var Specifies the target read-modify-write (Rmw) variable (in FORTRAN) or its address (in C). The value of this parameter cannot be NULL (in C) or LAPI ADDR NULL (in FORTRAN).

Specifies the value that is passed in to the operation (*op*). This value cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

INPUT/OUTPUT

in val

prev tgt val Specifies the location at the origin in which the previous tgt var on the target task is

stored before the operation (op) is executed. The value of this parameter can be NULL (in

C) or **LAPI_ADDR_NULL** (in FORTRAN).

org_cntr Specifies the origin counter address (in C) or the origin counter (in FORTRAN). If

prev_tgt_val is set, the origin counter (org_cntr) is incremented when prev_tgt_val is returned to the origin side. If prev_tgt_val is not set, the origin counter (org_cntr) is

updated after the operation (op) is completed at the target side.

OUTPUT

Restrictions

LAPI statistics are not reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

```
1. To synchronize a data value between two tasks (with FETCH AND ADD):
          int local var;
          int *addr list;
          /* both tasks initialize local_var to a value
                                                              */
          /* local var addresses are exchanged and stored
                                                              */
          /* in addr list (using LAPI Address init).
                                                              */
          /* addr_list[tgt] now contains the address of
                                                              */
          /* local var on tgt
          /* add value to local var on some task
                                                               */
          /* use LAPI to add value to local var on remote task */
          LAPI Rmw(hndl, FETCH AND ADD, tgt, addr list[tgt],
                   value, prev_tgt_val, &org_cntr);
          /* local var on the remote task has been increased
                                                               */
          /* by value. prev tgt val now contains the value
          /* of local_var on remote task before the addition
                                                               */
2. To synchronize a data value between two tasks (with SWAP):
   {
         int local var;
         int *addr list;
         /* local var addresses are exchanged and stored */
         /* in addr_list (using LAPI_Address_init).
                                                          */
         /* addr list[tgt] now contains the address of
         /* local var on tgt.
         /* local_var is assigned some value
         /* assign local var to local var on remote task */
         LAPI_Rmw(hndl, SWAP, tgt, addr_list[tgt],
                  local_var, prev_tgt_val, &org_cntr);
         /* local_var on the remote task is now equal to */
         /* local_var on the local task. prev_tgt_val now */
         /* contains the value of local_var on the remote */
         /* task before the swap.
```

3. To conditionally swap a data value (with **COMPARE_AND_SWAP**):

```
{
      int local_var;
      int *addr list;
      int in val[2];
      /* local var addresses are exchanged and stored
      /* in addr_list (using LAPI_Address_init).
      /* addr_list[tgt] now contains the address of
                                                              */
      /* local var on tgt.
      /* if local var on remote task is equal to comparator, */
      /* assign value to local_var on remote task
      in val[0] = comparator;
      in val[1] = value;
      LAPI Rmw(hndl, COMPARE AND SWAP, tgt, addr list[tgt],
               in val, prev tgt val, &org cntr);
      /* local var on the remote task is now in val[1] if it */
      /* had previously been equal to in val[0]. If the swap */
      /* was performed, prev_tgt_val now contains TRUE;
      /* otherwise, it contains FALSE.
}
```

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the *hndl* passed in is not valid (not initialized or in

terminated state).

Indicates that the *in_val* pointer is NULL (in C) or that the value of *in_val* LAPI_ERR_IN_VAL_NULL

is LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_RMW_OP Indicates that op is not valid.

LAPI_ERR_TGT Indicates that the tgt passed in is outside the range of tasks defined in the

LAPI_ERR_TGT_PURGED Indicates that the subroutine returned early because LAPI_Purge_totask()

was called.

LAPI_ERR_TGT_VAR_NULL Indicates that the tgt_var address is NULL (in C) or that the value of

tgt_var is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Address_init, LAPI_Getcntr, LAPI_Qenv, LAPI_Rmw64, LAPI_Setcntr, LAPI_Waitcntr, LAPI_Xfer

LAPI_Rmw64 Subroutine

Purpose

Provides data synchronization primitives for 64-bit applications.

Library

Availability Library (liblapi r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Rmw64(hndl, op, tgt, tgt var, in val, prev tgt val, org cntr)
lapi handle t hndl;
Rmw_ops_t op;
uint tgt;
long long *tgt var;
long long *in_val;
long long *prev tgt val;
lapi cntr t *org cntr;
```

FORTRAN Syntax

```
include 'lapif.h'
LAPI_RMW64(hndl, op, tgt, tgt var, in val, prev tgt val, org cntr, ierror)
INTEGER hndl
INTEGER op
\mathbf{INTEGER}\ tgt
INTEGER (KIND=LAPI ADDR TYPE) :: tqt var
INTEGER (KIND=LAPI LONG LONG TYPE) :: in val, prev tgt val
TYPE (LAPI_CNTR_T) :: org cntr
INTEGER ierror
```

Description

Type of call: point-to-point communication (non-blocking)

This subroutine is the 64-bit version of LAPI_Rmw. It is used to synchronize two independent pieces of 64-bit data, such as two tasks sharing a common data structure. The operation is performed at the target task (tgt) and is atomic. The operation takes an input value (in_val) from the origin and performs one of four operations (op) on a variable (tat var) at the target (tat), and then replaces the target variable (tgt_var) with the results of the operation (op). The original value (prev_tgt_val) of the target variable (tat_var) is returned to the origin.

The operations (op) are performed over the context referred to by hndl. The outcome of the execution of these calls is as if the following code was executed atomically:

```
*prev tgt val = *tgt var;
*tgt var
             = f(*tgt var, *in val);
where:
f(a,b) = a + b for FETCH_AND_ADD
f(a,b) = a \mid b \text{ for } FETCH \text{ AND } OR \text{ (bitwise or)}
```

```
f(a,b) = b for SWAP
```

For **COMPARE_AND_SWAP**, *in_val* is treated as a pointer to an array of two integers, and the *op* is the following atomic operation:

```
if(*tgt_var == in_val[0]) {
   *prev_tgt_val = TRUE;
   *tgt_var = in_val[1];
} else {
   *prev_tgt_val = FALSE;
}
```

This subroutine can also be used on a 32-bit processor.

All **LAPI_Rmw64** calls are non-blocking. To test for completion, use the **LAPI_Getcntr** and **LAPI_Waitcntr** subroutines. **LAPI_Rmw64** does not include a target counter (*tgt_cntr*), so **LAPI_Rmw64** calls do not provide any indication of completion on the target task (*tgt*).

Parameters

INPUT

hndl Specifies the LAPI handle.

op Specifies the operation to be performed. The valid operations are:

COMPARE_AND_SWAP

FETCH AND ADD

FETCH_AND_OR

SWAP

tgt Specifies the task ID of the target task where the read-modify-write (Rmw64) variable

resides. The value of this parameter must be in the range 0 <= tgt < NUM_TASKS.

tgt_var Specifies the target read-modify-write (Rmw64) variable (in FORTRAN) or its address (in

C). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN).

in_val Specifies the value that is passed in to the operation (op). This value cannot be NULL (in

C) or LAPI ADDR NULL (in FORTRAN).

INPUT/OUTPUT

prev tqt val Specifies the location at the origin in which the previous tqt var on the target task is

stored before the operation (op) is executed. The value of this parameter can be NULL (in

C) or LAPI ADDR NULL (in FORTRAN).

org_cntr Specifies the origin counter address (in C) or the origin counter (in FORTRAN). If

prev_tgt_val is set, the origin counter (org_cntr) is incremented when prev_tgt_val is returned to the origin side. If prev_tgt_val is not set, the origin counter (org_cntr) is

updated after the operation (op) is completed at the target side.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

```
1. To synchronize a data value between two tasks (with FETCH_AND_ADD):
         long long local var;
         long long *addr list;
         /* both tasks initialize local var to a value
         /* local_var addresses are exchanged and stored
         /* in addr list (using LAPI Address init64)
                                                              */
         /* addr list[tgt] now contains address of
         /* local var on tgt
         /* add value to local_var on some task
         /* use LAPI to add value to local var on remote task */
         LAPI_Rmw64(hndl, FETCH_AND_ADD, tgt, addr_list[tgt],
                    value, prev_tgt_val, &org_cntr);
         /* local var on remote task has been increased
                                                              */
         /* by value. prev_tgt_val now contains value of
         /* local_var on remote task before the addition
2. To synchronize a data value between two tasks (with SWAP):
   {
         long long local_var;
         long long *addr list;
         /* local var addresses are exchanged and stored
         /* in addr_list (using LAPI_Address_init64).
         /* addr_list[tgt] now contains the address of
         /* local_var on tgt.
         /* local_var is assigned some value
         /* assign local var to local var on the remote task
         LAPI Rmw64(hndl, SWAP, tgt, addr list[tgt],
                    local_var, prev_tgt_val, &org_cntr);
         /* local_var on the remote task is now equal to local var */
         /* on the local task. prev tgt val now contains the value */
         /* of local var on the remote task before the swap.
3. To conditionally swap a data value (with COMPARE_AND_SWAP):
   {
         long long local var;
         long long *addr list;
         long long in val[2];
         /* local var addresses are exchanged and stored
         /* in addr_list (using LAPI_Address_init64).
         /* addr list[tgt] now contains the address of
         /* local var on tgt.
```

```
/* if local_var on remote_task is equal to comparator, */
     /* assign value to local_var on the remote task
      in val[0] = comparator;
      in val[1] = value;
      LAPI Rmw64(hndl, COMPARE_AND_SWAP, tgt, addr_list[tgt],
                 in val, prev tgt val, &org cntr);
      /* local var on remote task is now in val[1] if it
      /* had previously been equal to in_val[0]. If the
      /* swap was performed, prev_tgt_val now contains
      /* TRUE; otherwise, it contains FALSE.
}
```

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the *hndl* passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_IN_VAL_NULL Indicates that the *in_val* pointer is NULL (in C) or that the value of *in_val*

is LAPI ADDR NULL (in FORTRAN).

LAPI_ERR_RMW_OP Indicates that op is not valid.

LAPI_ERR_TGT Indicates that the tgt passed in is outside the range of tasks defined in the

LAPI ERR TGT PURGED Indicates that the subroutine returned early because LAPI Purge totask()

was called.

LAPI ERR TGT VAR NULL Indicates that the tgt_var address is NULL (in C) or that the value of

tgt_var is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI Address init64, LAPI Getcntr, LAPI Qenv, LAPI Rmw, LAPI Setcntr, LAPI_Waitcntr, LAPI_Xfer

LAPI_Senv Subroutine

Purpose

Used to set a runtime variable.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapif.h>
int LAPI_Senv(hndl, query, set_val)
lapi_handle_t hndl;
lapi_query_t query;
int set_val;
```

FORTRAN Syntax

```
include 'lapif.h'
LAPI_SENV(hndl, query, set_val, ierror)
INTEGER hndl
INTEGER query
INTEGER set_val
INTEGER ierror
```

Description

Type of call: local queries

Use this subroutine to set runtime attributes for a specific LAPI instance. In C, the **lapi_query_t** enumeration defines the attributes that can be set at runtime. These attributes are defined explicitly in FORTRAN. See **LAPI_Qenv** for more information.

You can use LAPI_Senv to set these runtime attributes: ACK_THRESHOLD, ERROR_CHK, INTERRUPT SET, and TIMEOUT.

Parameters

INPUT

hndl Specifies the LAPI handle.

query Specifies the type of query that you want to set. In C, the values for query are defined by

the lapi_query_t enumeration in lapi.h. In FORTRAN, these values are defined explicitly

in the 32-bit version and the 64-bit version of lapif.h.

set_val Specifies the integer value of the query that you want to set.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

The following values can be set using LAPI_Senv:

```
boolean toggle;
LAPI Senv(hndl, ERROR CHK, toggle);
/* Indicates whether LAPI should perform error checking. If set, LAPI */
/* calls will perform bounds-checking on parameters. Error checking
/* is disabled by default.
INTERRUPT SET:
boolean toggle;
LAPI_Senv(hndl, INTERRUPT_SET, toggle);
/* Determines whether LAPI will respond to interrupts. If interrupts
/* are disabled, LAPI will poll for message completion.
/* toggle==True will enable interrupts, False will disable.
                                                                       */
/* Interrupts are enabled by default.
TIMEOUT:
int value;
LAPI_Senv(hndl, TIMEOUT, value);
/* LAPI will time out on a communication if no response is received
/* within timeout seconds. Valid range is (10 <= timeout <= 86400).
/* 86400 seconds = 24 hours. Default value is 900 (15 minutes).
```

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

Indicates that the hndl passed in is not valid (not initialized or in LAPI ERR HNDL INVALID

terminated state).

LAPI ERR QUERY TYPE Indicates the query passed in is not valid.

LAPI_ERR_SET_VAL Indicates the set_val pointer is not in valid range.

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI Qenv

LAPI_Setcntr Subroutine

Purpose

Used to set a counter to a specified value.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Setcntr(hndl, cntr, val)
lapi handle t hndl;
lapi_cntr_t *cntr;
             val;
```

FORTRAN Syntax

```
include 'lapif.h'
LAPI_SETCNTR(hndl, cntr, val, ierror)
```

```
INTEGER hndl
TYPE (LAPI CNTR T) :: cntr
INTEGER val
INTEGER ierror
```

Description

Type of call: Local counter manipulation

This subroutine sets cntr to the value specified by val. Because the LAPI_Getcntr/LAPI_Setcntr sequence cannot be made atomic, you should only use LAPI_Setcntr when you know there will not be any competing operations.

Parameters

INPUT

hndl Specifies the LAPI handle.

val Specifies the value to which the counter needs to be set.

INPUT/OUTPUT

cntr

Specifies the address of the counter to be set (in C) or the counter structure (in FORTRAN). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

OUTPUT

ierror

Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are not reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

```
To initialize a counter for use in a communication API call:
    lapi cntr t  my tgt cntr, *tgt cntr array;
                  initial value, expected value, current value;
    lapi_handle_t hndl;
    * Note: the code below is executed on all tasks
    /* initialize, allocate and create structures */
    initial value = 0;
    expected_value = 1;
    /* set the cntr to zero */
    LAPI Setcntr(hndl, &my tgt cntr, initial value);
    /* set other counters */
    /* exchange counter addresses, LAPI Address init synchronizes */
    LAPI Address_init(hndl, &my_tgt_cntr, tgt_cntr_array);
    /* more address exchanges */
```

```
/* Communication calls using my tgt cntr */
LAPI Put(...., tgt cntr array[tgt], ....);
/* Wait for counter to reach value */
for (;;) {
    LAPI_Getcntr(hndl, &my_tgt_cntr, &current_value);
    if (current_value >= expected value) {
       break; /* out of infinite loop */
    } else {
       LAPI Probe(hnd1);
}
/* Quiesce/synchronize to ensure communication using our counter is done */
LAPI Gfence(hnd1);
/* Reset the counter */
LAPI Setcntr(hndl, &my tgt cntr, initial value);
* Synchronize again so that no other communication using the counter can
 * begin from any other task until we're all finished resetting the counter.
LAPI_Gfence(hnd1);
/* More communication calls */
```

Return Values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL Indicates that the cntr value passed in is NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Getcntr, LAPI_Waitcntr

LAPI_Setcntr_wstatus Subroutine

Purpose

Used to set a counter to a specified value and to set the associated destination list array and destination status array to the counter.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
int LAPI_Setcntr_wstatus(hndl, cntr, num_dest, dest_list, dest_status)
lapi_handle_t hndl;
lapi_cntr_t *cntr;
int num_dest;
uint *dest_list;
int *dest_status;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_SETCNTR_WSTATUS(hndl, cntr, num_dest, dest_list, dest_status, ierror)
INTEGER hndl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER num_dest
INTEGER dest_list(*)
INTEGER dest_status
INTEGER ierror
```

Description

Type of call: recovery

This subroutine sets *cntr* to **0**. Use **LAPI_Setcntr_wstatus** to set the associated destination list array (*dest_list*) and destination status array (*dest_status*) to the counter. Use a corresponding **LAPI_Nopoll_wait** call to access these arrays. These arrays record the status of a task from where the thread calling **LAPI Nopoll wait()** is waiting for a response.

The return values for dest status are:

LAPI_MSG_INITIAL The task is purged or is not received.

LAPI_MSG_RECVD The task is received.

LAPI_MSG_PURGED The task is purged, but not received.

LAPI_MSG_PURGED_RCVD The task is received and then purged.

LAPI_MSG_INVALID Not valid; the task is already purged.

Note: To use this subroutine, the *lib_vers* field in the **lapi_info_t** structure must be set to **L2_LIB** or **LAST_LIB**.

Parameters

INPUT

hndl Specifies the LAPI handle.

num_dest Specifies the number of tasks in the destination list.

dest_list Specifies an array of destinations waiting for this counter update. If the value of this

parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), no status is returned to

the user.

INPUT/OUTPUT

cntr Specifies the address of the counter to be set (in C) or the counter structure (in

FORTRAN). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN).

OUTPUT

dest status Specifies an array of status that corresponds to dest_list. The value of this parameter can

be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

Use of this subroutine is not recommmended on a system that is running Parallel Environment (PE).

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL Indicates that the cntr value passed in is NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_RET_PTR_NULL Indicates that the value of dest_status is NULL in C (or

LAPI_ADDR_NULL in FORTRAN), but the value of *dest_list* is not NULL

in C (or LAPI_ADDR_NULL in FORTRAN).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Getcntr, LAPI_Nopoll_wait, LAPI_Purge_totask, LAPI_Setcntr

LAPI Term Subroutine

Purpose

Terminates and cleans up a LAPI context.

Library

Availability Library (liblapi_r.a)

C Syntax

#include <lapi.h>

int LAPI Term(hndl) lapi_handle_t hndl;

FORTRAN Syntax

include 'lapif.h'

LAPI TERM(hndl, ierror) INTEGER hndl **INTEGER** ierror

Description

Type of call: local termination

Use this subroutine to terminate the LAPI context that is specified by hndl. Any LAPI notification threads that are associated with this context are terminated. An error occurs when any LAPI calls are made using hndl after LAPI_Term is called.

A DGSP that is registered under that LAPI handle remains valid even after LAPI_Term is called on hndl.

Parameters

INPUT

hndl Specifies the LAPI handle.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are not reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

To terminate a LAPI context (represented by hndl): LAPI Term(hndl);

Return Values

LAPI SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

Location

/usr/lib/liblapi r.a

Related Information

Subroutines: LAPI_Init, LAPI_Purge_totask, LAPI_Resume_totask

LAPI Util Subroutine

Purpose

Serves as a wrapper function for LAPI registration, reservation, and other data gather-scatter functions.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>
int LAPI Util(hndl, util cmd)
lapi handle t hndl;
lapi_util_t *util cmd;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_UTIL(hndl, util_cmd, ierror)
INTEGER hndl
TYPE (LAPI_UTIL_T) :: util_cmd
INTEGER ierror
```

Description

Type of call: DGSP utilities

This subroutine is used for a number of operations selected by the command type value in the beginning of the command structure. The **lapi_util_t** structure is defined as:

The enumerated type lapi util type t has these values:

Table 1. lapi_util_type_t types

Value of Util_type	Union member as interpreted by LAPI_Util
LAPI_REGISTER_DGSP	lapi_reg_dgsp_t
LAPI_RESERVE_DGSP	lapi_resv_dgsp_t
LAPI_UNRESERVE_DGSP	lapi_dref_dgsp_t
LAPI_REG_DDM_FUNC	lapi_reg_ddm_t
LAPI_ADD_UDP_DEST_PORT	lapi_add_udp_port_t
LAPI_DGSP_PACK	lapi_pack_dgsp_t
LAPI_DGSP_UNPACK	lapi_unpack_dgsp_t

hndl is not checked for command type **LAPI_REGISTER_DGSP**, **LAPI_RESERVE_DGSP**, or **LAPI_UNRESERVE_DGSP**.

LAPI REGISTER DGSP

This command registers a user-constructed LAPI DGSP program using the <code>lapi_reg_dgsp_t</code> command structure. To register a LAPI DGSP program, <code>lapi_dgsp_descr_t</code> idgsp must be passed in. LAPI returns a handle (<code>lapi_dg_handle_t</code> dgsp_handle) to use for all future LAPI calls. Note that the dgsp_handle that is returned by a register operation is identified as a <code>lapi_dg_handle_t</code>, which is the appropriate type for <code>LAPI_Xfer</code> and <code>LAPI_Util</code> calls that take a DGSP. This returned dgsp_handle is also defined to be castable to a pointer to a <code>lapi_dgsp_descr_t</code> for those situations where the LAPI user requires read-only access to information that is contained in the cached DGSP. The register operation delivers a DGSP to LAPI for use in future message send, receive, pack, and unpack operations. LAPI creates its own copy of the DGSP and protects it by reference count. All internal LAPI operations that depend on a DGSP cached in LAPI ensure the preservation of the DGSP by incrementing the reference count when they begin a dependency on the DGSP and decrementing the count when that dependency ends. A DGSP, once registered, can be used from any LAPI instance. <code>LAPI_Term</code> does not discard any DGSPs.

You can register a DGSP, start one or more LAPI operations using the DGSP, and then unreserve it with no concern about when the LAPI operations that depend on the DGSP will be done using it. See **LAPI_RESERVE_DGSP** and **LAPI_UNRESERVE_DGSP** for more information.

The DGSP that is built by the LAPI user and passed in to the LAPI_REGISTER_DGSP call using the dgsp parameter is normally discarded after LAPI makes and caches its own copy. Because DGSP creation is complex, user errors may occur, but extensive error checking at data transfer time would hurt performance. When developing code that creates DGSPs, extra validation can be invoked at the point of registration by setting the LAPI_VERIFY_DGSP environment variable. Any detected errors will be returned by LAPI Util. Any errors that exist and are not detected at registration time will cause problems during data transfer. Any errors detected during data transfer will be reported by an asynchronous error handler. A segmentation fault is one common symptom of a faulty DGSP. If multiple DGSPs are in use, the asynchronous error handler will not be able to identify which DGSP caused the error. For more information about asynchronous error handling, see LAPI_Init.

Table 2. The lapi_reg_dgsp_t fields

lapi_reg_dgsp_t field	lapi_reg_dgsp_t field type	lapi_reg_dgsp_t usage
Util_type	lapi_util_type_t	LAPI_REGISTER_DGSP
idgsp	lapi_dgsp_descr_t	IN - pointer to DGSP program
dgsp_handle	lapi_dg_handle_t	OUT - handle for a registered DGSP program
in_usr_func	lapi_usr_fcall_t	For debugging only
status	lapi_status_t	OUT - future support

LAPI UNRESERVE DGSP

An un-reserve operation decrements the user reference count allowing LAPI to free the DGSP if both external and internal reference counts are zero. Un-reserve uses a lapi dref dqsp t command structure. All operations that decrement a reference count will cause LAPI to check to see if the counts have both become 0 and if they have, dispose of the DGSP. Several internal LAPI activities increment and decrement a second reference count. The cached DGSP is disposable only when all activities, internal and external, which depend in it and use reference counting to preserve it have discharged their reference. The DGSP handle is passed to LAPI as a value parameter and LAPI does not nullify the caller's handle. It is your responsibility to not use this handle again because in doing an un-reserve operation, you have indicated that you no longer count on the handle remaining valid.

Table 3. The lapi_dref_dgsp_t fields

lapi_dref_dgsp_t field	lapi_dref_dgsp_t field type	lapi_dref_dgsp_t usage
Util_type	lapi_util_type_t	LAPI_UNRESERVE_DGSP
dgsp_handle	lapi_dg_handle_t	OUT - handle for a registered DGSP program
in_usr_func	lapi_usr_fcall_t	For debugging only
status	lapi_status_t	OUT - future support

LAPI_RESERVE_DGSP

A reserve operation is given a DGSP handle (which is also defined to be a pointer to a DGSP description already cached inside LAPI. This function is provided because a LAPI client may cache a LAPI DGSP handle for later use and needs to ensure the DGSP will not be discarded before the cached handle is used. The DGSP handle is defined to also be a structure pointer so client code can directly access information in the DGSP. Unless the client can be certain the DGSP will not be UNRESERVED by another thread while it is being accessed, the client should bracket the access window with its own RESERVE/UNRESERVE. The client is not to modify the cached DGSP but LAPI has no way to enforce

this. The reserve operation increments the user reference count, thus protecting the DGSP until un-reserve. This is needed because the thread that placed the reservation will expect to be able to use or examine the cached DGSP until it makes an un-reserve call which decrements the user reference count, even if the unreserve that matches the original register operation occurs within this window on some other thread. A reserve operation uses a lapi resv dgsp t command structure.

Table 4. The lapi_resv_dgsp_t fields

lapi_resv_dgsp_t field	lapi_resv_dgsp_t field type	lapi_resv_dgsp_t usage
Util_type	lapi_util_type_t	LAPI_RESERVE_DGSP
dgsp_handle	lapi_dg_handle_t	OUT - handle for a registered DGSP program
in_usr_func	lapi_usr_fcall_t	For debugging only
status	lapi_status_t	OUT - future support

LAPI REG DDM FUNC

This function:

- · is used to establish a callback function for processing data that is being scattered into a user buffer on the destination side.
- is used primarily for MPI Accumulate, but LAPI clients can provide any data distribution manager (DDM) function.
- works in conjunction with the DGSM CONTROL instruction.
- uses a lapi_reg_ddm_t command structure. Each call replaces the previous function pointer, if there was one.

The native LAPI user can install a callback without affecting the one MPI has registered for **MPI Accumulate.** The function prototype for the callback function is:

```
typedef long ddm_func_t (
                                /* return number of bytes processed */
        void
                                /* ptr to inbound data
                  *in.
                  *inout,
                               /* ptr to destination space
        void
                                                                        */
                              /* number of bytes inbound
/* CONTROL operand value
                                                                       */
        long
                  bytes,
        int
                                                                       */
                  operand.
        int
                                /* CONTROL operation value
                  operation
                                                                       */
);
```

A DDM function acts between the arrival of message data and the target buffer. The most common usage is to combine inbound data with data already in the target buffer. For example, if the target buffer is an array of integers and the incoming message consists of integers, the DDM function can be written to add each incoming integer to the value that is already in the buffer. The operand and operation fields of the DDM function allow one DDM function to support a range of operations with the CONTROL instruction by providing the appropriate values for these fields.

See RSCT for AIX 5L: LAPI Programming Guide for more information about DGSP programming.

Table 5. The lapi_reg_ddm_t fields

lapi_reg_ddm_t field	lapi_reg_ddm_t field type	lapi_reg_ddm_t usage
Util_type	lapi_util_type_t	LAPI_REG_DDM_FUNC
ddm_func	ddm_func_t *	IN - DDM function pointer
in_usr_func	lapi_usr_fcall_t	For debugging only
status	lapi_status_t	OUT - future support

LAPI_ADD_UDP_DEST_PORT

The command is used to update the UDP port information of the destination task. This is unavailable under POE runtime environment. This function can be used when the user has written user's own udp_hndlr and needs to support recovery of failed tasks. Update UDP port uses a lapi_add_udp_port_t command structure.

Table 6. The lapi_add_udp_port_t fields

lapi_add_udp_port_t field	lapi_add_udp_port_t field type	lapi_add_udp_port_t usage
Util_type	lapi_util_type_t	LAPI_ADD_UDP_DEST_PORT
tgt	uint	IN - destination task ID
udp_port	In - UDP port information for the target	
instance_no	uint IN - Instance number of UDP	
in_usr_func	lapi_usr_fcall_t	For debugging only
status	lapi_status_t OUT - future support	

LAPI DGSP PACK

This command is used to gather data to a pack buffer from a user buffer under control of a DGSP. A single buffer may be packed by a series of calls. The caller provides a "position" value initialized to the starting offset within the buffer. Each pack operation adjusts "position" so the next pack can pick up where the previous left off. Normally a series of pack operations will begin with "position" initialized to 0 but any offset is legitimate. There is no state carried from one pack to the next. Each pack operation starts at the beginning of the DGSP it is passed. Pack DGSP uses a lapi pack dgsp t command structure.

Table 7. The lapi_pack_dgsp_t fields

lapi_pack_dgsp_t field	lapi_pack_dgsp_t field type	lapi_pack_dgsp_t usage
Util_type	lapi_util_type_t LAPI_DGSP_PACK	
dgsp_handle	lapi_dg_handle_t	OUT - handle for a registered DGSP program
in_buf	void *	IN - source buffer to pack
bytes	ulong	IN - number of bytes to pack
out_buf	void * OUT - output buffer for pack	
out_size	ulong IN - output buffer size in bytes	
position	ulong IN/OUT - current buffer offset	
in_usr_func	lapi_usr_fcall_t For debugging only	
status	lapi_status_t OUT - future support	

LAPI_DGSP_UNPACK

This command is used to scatter data from a packed buffer to a user buffer under control of a DGSP. A single buffer may be unpacked by a series of calls. The caller provides a "position" value initialized to the starting offset within the packed buffer. Each pack operation adjusts "position" so the next unpack can pick up where the previous left off. Normally a series of unpack operations will begin with "position" initialized to 0 but any offset is legitimate. There is no state carried from one unpack to the next. Each unpack operation starts at the beginning of the DGSP it is passed. Unpack DGSP uses a lapi unpack dgsp t command structure.

Table 8. The lapi_unpack_dgsp_t fields

lapi_unpack_dgsp_t field	lapi_unpack_dgsp_t field type	lapi_unpack_dgsp_t usage
Util_type	lapi_util_type_t	LAPI_DGSP_UNPACK

Table 8. The lapi_unpack_dgsp_t fields (continued)

lapi_unpack_dgsp_t field	lapi_unpack_dgsp_t field type lapi_unpack_dgsp_t usage	
dgsp_handle	lapi_dg_handle_t	OUT - handle for a registered DGSP program
buf	void *	IN - source buffer for unpack
in_size	ulong	IN - source buffer size in bytes
out_buf	void *	OUT - output buffer for unpack
bytes	ulong IN - number of bytes to unpack	
out_size	ulong IN - output buffer size in bytes	
position	ulong IN/OUT - current buffer offset	
in_usr_func	lapi_usr_fcall_t For debugging only	
status	lapi_status_t OUT - future support	

Parameters

INPUT

hndl Specifies the LAPI handle.

INPUT/OUTPUT

util cmd

Specifies the command type of the utility function.

OUTPUT

ierror

Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

1. To create and register a DGSP:

```
{
     ** DGSP code array. DGSP instructions are stored
     ** as ints (with constants defined in lapi.h for
     \ensuremath{\mbox{**}} the number of integers needed to store each
     ** instruction). We will have one COPY and one ITERATE
     ** instruction in our DGSP. We use LAPI's constants
     ** to allocate the appropriate storage.
     */
     int code[LAPI_DGSM_COPY_SIZE+LAPI_DGSM_ITERATE_SIZE];
     /* DGSP description */
     lapi dgsp descr t dgsp d;
     ** Data structure for the xfer call.
     lapi xfer t xfer struct;
     /* DGSP data structures */
     lapi dgsm copy t *copy p;
                                     /* copy instruction
                                     /* iterate instruction */
     lapi dgsm iterate t *iter p;
                           *code ptr; /* code pointer
```

```
/* constant for holding code array info */
int
                      code less iterate size;
/* used for DGSP registration */
lapi_reg_dgsp_t
                  reg util;
** Set up dgsp description
/* set pointer to code array */
dgsp_d.code = &code[0];
/* set size of code array */
dgsp_d.code_size = LAPI_DGSM_COPY_SIZE + LAPI_DGSM_ITERATE_SIZE;
/* not using DGSP gosub instruction */
dgsp_d.depth = 1;
** set density to show internal gaps in the
** DGSP data layout
dgsp_d.density = LAPI_DGSM_SPARSE;
/* transfer 4 bytes at a time */
dgsp d.size = 4;
/* advance the template by 8 for each iteration */
dgsp_d.extent = 8;
** ext specifies the memory 'footprint' of
** data to be transferred. The lext specifies
** the offset from the base address to begin
** viewing the data. The rext specifies the
** length from the base address to use.
*/
dgsp d.lext = 0;
dgsp d.rext = 4;
/* atom size of 0 lets LAPI choose the packet size */
dgsp d.atom size = 0;
** set up the copy instruction
copy_p = (lapi_dgsm_copy_t *)(dgsp_d.code);
copy_p->opcode = LAPI_DGSM COPY;
/* copy 4 bytes at a time */
copy_p->bytes = (long) 4;
/* start at offset 0 */
copy p->offset = (long) 0;
/* set code pointer to address of iterate instruction */
code_less_iterate_size = dgsp_d.code_size - LAPI_DGSM_ITERATE_SIZE;
code_ptr = ((int *)(code))+code_less_iterate_size;
** Set up iterate instruction
*/
iter p = (lapi dgsm iterate t *) code ptr;
iter p->opcode = LAPI DGSM ITERATE;
iter p->iter loc = (-code less iterate size);
```

```
/* Set up and do DGSP registration */
         reg util.Util type = LAPI REGISTER DGSP;
         reg_util.idgsp = &dgsp_d;
         LAPI_Util(hndl, (lapi_util_t *)&reg_util);
         ** LAPI returns a usable DGSP handle in
         ** reg_util.dgsp_handle
         ** Use this handle for subsequent reserve/unreserve
         ** and Xfer calls. On the receive side, this
         ** handle can be returned by the header handler using the
         ** return_info_t mechanism. The DGSP will then be used for
         ** scattering data.
2. To reserve a DGSP handle:
   {
         reg util.dgsp handle = dgsp handle;
         ** dgsp handle has already been created and
         ** registered as in the above example
         reg util.Util type = LAPI RESERVE DGSP;
         LAPI_Util(hndl, (lapi_util_t *)&reg_util);
         /*
         ** LAPI's internal reference count to dgsp handle
         ** will be incremented. DGSP will
         ** remain available until an unreserve is
         ** done for each reserve, plus one more for
         ** the original registration.
3. To un-reserve a DGSP handle:
    {
         reg_util.dgsp_handle = dgsp_handle;
         ** dgsp handle has already created and
         ** registered as in the above example, and
         ** this thread no longer needs it.
         reg util.Util type = LAPI UNRESERVE DGSP;
         LAPI_Util(hndl, (lapi_util_t *)&reg_util);
         ** An unreserve is required for each reserve,
         ** plus one more for the original registration.
   }
```

Return Values

LAPI SUCCESS

Indicates that the function call completed successfully.

LAPI_ERR_DGSP Indicates that the DGSP that was passed in is NULL (in C) or

LAPI ADDR NULL (in FORTRAN) or is not a registered DGSP.

LAPI_ERR_DGSP_ATOM Indicates that the DGSP has an *atom_size* that is less than **0** or greater

than MAX_ATOM_SIZE.

LAPI_ERR_DGSP_BRANCH Indicates that the DGSP attempted a branch that fell outside the code

array. (Returned only in validation mode.)

LAPI_ERR_DGSP_COPY_SZ Error returned with DGSP validation turned on when MCOPY block < 0 or

COPY instruction with bytes < **0**. (Returned only in validation mode.)

LAPI_ERR_DGSP_FREETry to free dgsp which is not valid or is no longer registered. (There

should be one LAPI_UNRESERVE_DGSP to close the

LAPI_REGISTER_DGSP plus one LAPI_UNRESERVE_DGSP for each

LAPI_RESERVE_DGSP.)

LAPI_ERR_DGSP_OPCODE Indicates that the DGSP op-code is not valid. (Returned only in validation

mode.)

LAPI_ERR_DGSP_STACK Indicates that the DGSP has greater GOSUB depth than the allocated

stack supports. Stack allocation is specified by the dgsp->depth member.

(Returned only in validation mode.)

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_MEMORY_EXHAUSTED

Indicates that LAPI is unable to obtain memory from the system.

LAPI_ERR_UDP_PORT_INFO

Indicates that the *udp_port* information pointer is NULL (in C) or that the

value of *udp_port* is **LAPI_ADDR_NULL** (in FORTRAN).

LAPI_ERR_UTIL_CMD Indicates that the command type is not valid.

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI Init, LAPI Xfer

LAPI_Waitcntr Subroutine

Purpose

Waits until a specified counter reaches the value specified.

Library

Availability Library (liblapi r.a)

C Syntax

```
#include <lapi.h>
```

```
int LAPI_Waitcntr(hndl, cntr, val, cur_cntr_val)
lapi_handle_t hndl;
lapi_cntr_t *cntr;
int val;
int *cur cntr val;
```

FORTRAN Syntax

```
include 'lapif.h'
LAPI WAITCNTR(hndl, cntr, val, cur cntr val, ierror)
INTEGER hndl
TYPE (LAPI CNTR T) :: cntr
INTEGER va\overline{l}
INTEGER cur cntr val
INTEGER ierror
```

Description

Type of call: local progress monitor (blocking)

This subroutine waits until cntr reaches or exceeds the specified val. Once cntr reaches val, cntr is decremented by the value of val. In this case, "decremented" is used (as opposed to "set to zero") because cntr could have contained a value that was greater than the specified val when the call was made. This call may or may not check for message arrivals over the LAPI context hndl. The cur_cntr_val variable is set to the current counter value.

Parameters

INPUT

Specifies the LAPI handle. hndl

val Specifies the value the counter needs to reach.

INPUT/OUTPUT

cntr Specifies the counter structure (in FORTRAN) to be waited on or its address (in C). The

value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

OUTPUT

Specifies the integer value of the current counter. This value can be NULL (in C) or cur_cntr_val

LAPI_ADDR_NULL (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

To wait on a counter to reach a specified value:

```
{
      int
                  val;
                  cur cntr val;
      lapi cntr t some cntr;
     LAPI Waitcntr(hndl, &some cntr, val, &cur cntr val);
      /* Upon return, some cntr has reached val */
```

Return Values

LAPI_SUCCESS

Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL Indicates that the *cntr* pointer is NULL (in C) or that the value of *cntr* is

LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID Indicates that the hndl passed in is not valid (not initialized or in

terminated state).

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Amsend, LAPI_Amsendv, LAPI_Get, LAPI_Getcntr, LAPI_Getv, LAPI_Put, LAPI Putv, LAPI Rmw, LAPI Rmw64, LAPI Setcntr, LAPI Xfer

LAPI Xfer Subroutine

Purpose

Serves as a wrapper function for LAPI data transfer functions.

Library

Availability Library (liblapi_r.a)

C Syntax

FORTRAN Syntax

```
include 'lapif.h'

LAPI_XFER(hndl, xfer_cmd, ierror)
INTEGER hndl
TYPE (fortran_xfer_type) :: xfer_cmd
INTEGER ierror
```

Description

Type of call: point-to-point communication (non-blocking)

The LAPI_Xfer subroutine provides a superset of the functionality of these subroutines: LAPI_Amsend, LAPI_Amsendv, LAPI_Putv, LAPI_Get, LAPI_Getv, and LAPI_Rmw. In addition, LAPI_Xfer provides data gather-scatter program (DGSP) messages transfer.

In C, the **LAPI_Xfer** command is passed a pointer to a union. It examines the first member of the union, **Xfer_type**, to determine the transfer type, and to determine which union member was passed. **LAPI_Xfer** expects every field of the identified union member to be set. It does not examine or modify any memory outside of the identified union member. **LAPI_Xfer** treats all union members (except **status**) as read-only data.

This subroutine provides the following functions:

- The remote address fields are expanded to be of type lapi_long_t, which is long enough for a 64-bit address. This allows a 32-bit task to send data to 64-bit addresses, which may be important in client/server programs.
- LAPI_Xfer allows the origin counter to be replaced with a send completion callback.
- LAPI_Xfer is used to transfer data using LAPI's data gather-scatter program (DGSP) interface.

The **lapi_xfer_t** structure is defined as:

```
typedef union {
        lapi_xfer_type_t
                          Xfer_type;
        lapi get t
                           Get;
        lapi am t
                           Am:
        lapi rmw t
                           Rmw;
        lapi put t
                           Put;
        lapi getv t
                           Getv;
        lapi_putv_t
                           Putv;
        lapi_amv_t
                           Amv;
        lapi amdgsp t
                           Dgsp;
} lapi xfer t;
```

Though the **lapi_xfer_t** structure applies only to the C version of **LAPI_Xfer**, the following tables include the FORTRAN equivalents of the C datatypes.

Table 9 list the values of the **lapi_xfer_type_t** structure for C and the explicit *Xfer_type* values for FORTRAN.

Table 9. LAPI_Xfer structure types

Value of Xfer_type (C or FORTRAN)	Union member as interpreted by LAPI_Xfer (C)	Value of fortran_xfer_type (FORTRAN)
LAPI_AM_XFER	lapi_am_t	LAPI_AM_T
LAPI_AMV_XFER	lapi_amv_t	LAPI_AMV_T
LAPI_DGSP_XFER	lapi_amdgsp_t	LAPI_AMDGSP_T
LAPI_GET_XFER	lapi_get_t	LAPI_GET_T
LAPI_GETV_XFER	lapi_getv_t	LAPI_GETV_T
LAPI_PUT_XFER	lapi_put_t	LAPI_PUT_T
LAPI_PUTV_XFER	lapi_putv_t	LAPI_PUTV_T
LAPI_RMW_XFER	lapi_rmw_t	LAPI_RMW_T

lapi am t details

Table 10 on page 565 shows the correspondence among the parameters of the **LAPI_Amsend** subroutine, the fields of the C **lapi_am_t** structure and their datatypes, and the equivalent FORTRAN datatypes. The **lapi_am_t** fields are listed in Table 10 on page 565 in the order that they occur in the **lapi_xfer_t** structure.

Table 10. LAPI_Amsend and lapi_am_t equivalents

lapi_am_t field name (C)	lapi_am_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Amsend parameter
Xfer_type	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C
			LAPI_Xfer value in FORTRAN: LAPI_AM_XFER
flags	int	INTEGER(KIND = 4)	none
			LAPI_Xfer parameter in FORTRAN: flags
tgt	uint	INTEGER(KIND = 4)	tgt
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN: pad
hdr_hdl	lapi_long_t	INTEGER(KIND = 8)	hdr_hdl
uhdr_len	uint	INTEGER(KIND = 4)	uhdr_len
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (64-bit): <i>pad2</i>
uhdr	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	uhdr
udata	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	udata
udata_len	ulong	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	udata_len
shdlr	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit)	none
		INTEGER(KIND = 8) (64-bit)	LAPI_Xfer parameter in FORTRAN: shdlr
sinfo	void *	INTEGER(KIND = 4) (32-bit)	none
		INTEGER(KIND = 8) (64-bit)	LAPI_Xfer parameter in FORTRAN: <i>sinfo</i>
tgt_cntr	lapi_long_t	INTEGER(KIND = 8)	tgt_cntr
org_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_cntr
cmpl_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	cmpl_cntr

When the origin data buffer is free to be used, the pointer to the send completion handler (shdlr) is called with the send completion data (sinfo) if shdlr is not a NULL pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). Otherwise, the behavior is identical to that of **LAPI_Amsend**.

lapi_amv_t details

Table 11 on page 566 shows the correspondence among the parameters of the LAPI_Amsendv subroutine, the fields of the C lapi_amv_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_amv_t fields are listed in Table 11 on page 566 in the order that they occur in the lapi_xfer_t structure.

Table 11. LAPI_Amsendv and lapi_amv_t equivalents

lapi_amv_t field name (C)	lapi_amv_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Amsendv parameter
Xfer_type	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C
			LAPI_Xfer value in FORTRAN: LAPI_AMV_XFER
flags	int	INTEGER(KIND = 4)	none
			LAPI_Xfer parameter in FORTRAN: flags
tgt	uint	INTEGER(KIND = 4)	tgt
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN: pad
hdr_hdl	lapi_long_t	INTEGER(KIND = 8)	hdr_hdl
uhdr_len	uint	INTEGER(KIND = 4)	uhdr_len
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (64-bit): pad2
uhdr	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	uhdr
shdlr	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit)	none
		INTEGER(KIND = 8) (64-bit)	LAPI_Xfer parameter in FORTRAN: shdlr
sinfo	void *	INTEGER(KIND = 4) (32-bit)	none
		INTEGER(KIND = 8) (64-bit)	LAPI_Xfer parameter in FORTRAN: sinfo
org_vec	lapi_vec_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_vec
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (32-bit): pad2
tgt_cntr	lapi_long_t	INTEGER(KIND = 8)	tgt_cntr
org_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_cntr
cmpl_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	cmpl_cntr

lapi_amdgsp_t details

Table 12 shows the correspondence among the fields of the C lapi_amdgsp_t structure and their datatypes, how they are used in LAPI_Xfer, and the equivalent FORTRAN datatypes. The lapi_amdgsp_t fields are listed in Table 12 in the order that they occur in the <code>lapi_xfer_t</code> structure.

Table 12. The lapi_amdgsp_t fields

lapi_amdgsp_t field name (C)		Equivalent FORTRAN datatype	LAPI_Xfer usage
Xfer_type	lapi_xfer_type_t	INTEGER(KIND = 4)	LAPI_DGSP_XFER

Table 12. The lapi_amdgsp_t fields (continued)

lapi_amdgsp_t field name (C)	lapi_amdgsp_t field type (C)	Equivalent FORTRAN datatype	LAPI_Xfer usage
flags	int	INTEGER(KIND = 4)	This field allows users to specify directives or hints to LAPI. If you do not want to use any directives or hints, you must set this field to 0 . See "The lapi_amdgsp_t flags field" for more information.
tgt	uint	INTEGER(KIND = 4)	target task
none	none	INTEGER(KIND = 4)	pad (padding alignment for FORTRAN only)
hdr_hdl	lapi_long_t	INTEGER(KIND = 8)	header handler to invoke at target
uhdr_len	uint	INTEGER(KIND = 4)	user header length (multiple of processor's doubleword size)
none	none	INTEGER(KIND = 4)	pad2 (padding alignment for 64-bit FORTRAN only)
uhdr	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	pointer to user header
udata	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	pointer to user data
udata_len	ulong	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	user data length
shdlr	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	send completion handler (optional)
sinfo	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	data pointer to pass to send completion handler (optional)
tgt_cntr	lapi_long_t	INTEGER(KIND = 8)	target counter (optional)
org_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	origin counter (optional)
cmpl_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	completion counter (optional)
dgsp	lapi_dg_handle_t	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	Handle of a registered DGSP
status	lapi_status_t	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	Status to return (future use)
none	none	INTEGER(KIND = 4)	pad3 (padding alignment for 64-bit FORTRAN only)

When the origin data buffer is free to be modified, the send completion handler (shdlr) is called with the send completion data (sinfo), if shdlr is not a NULL pointer (in C) or LAPI ADDR NULL (in FORTRAN).

See "Using lapi_am_dgsp_t for scatter-side DGSP processing" on page 568 for more information.

The lapi_amdgsp_t flags field: One or more flags can be set using the I (bitwise or) operator. User directives are always followed and could result in incorrect results if used improperly. Appropriate hints might improve performance, but they may be ignored by LAPI. Inappropriate hints might degrade performance, but they will not cause incorrect results.

The following directive flags are defined:

USE_TGT_VEC_TYPE

Instructs LAPI to use the vector type of the target vector (tgt vec). In other words, tat vec is to be interpreted as type lapi vec t; otherwise, it is interpreted as type lapi_lvec_t. The lapi_lvec_t type uses lapi_long_t. The lapi_vec_t type uses void * or long. Incorrect results will occur if one type is used in place of the other.

BUFFER BOTH CONTIGUOUS

Instructs LAPI to treat all data to be transferred as contiguous, which can improve performance. If this flag is set when non-contiguous data is sent, data will likely be corrupted.

The following hint flags are defined:

LAPI_NOT_USE_BULK_XFER

Instructs LAPI not to use bulk transfer, independent of the current setting

for the task.

LAPI USE BULK XFER Instructs LAPI to use bulk transfer, independent of the current setting for

the task.

If neither of these hint flags is set, LAPI will use the behavior defined for the task. If both of these hint flags are set, LAPI_NOT_USE_BULK_XFER will take precedence.

Using lapi am_dgsp_t for scatter-side DGSP processing: Beginning with AIX 5.2, LAPI allows additional information to be returned from the header handler through the use of the lapi return info t datatype. See RSCT for AIX 5L: LAPI Programming Guide for more information about lapi_return_info_t. In the case of transfer type lapi am dgsp t, this mechanism can be used to instruct LAPI to run a user DGSP to scatter data on the receive side.

To use this mechanism, pass a lapi_return_info_t * pointer back to LAPI through the msg_len member of the user header handler. The dasp handle member of the passed structure must point to a DGSP description that has been registered on the receive side. See LAPI_Util and RSCT for AIX 5L: LAPI Programming Guide for details on building and registering DGSPs.

lapi get t details

Table 13 shows the correspondence among the parameters of the LAPI Get subroutine, the fields of the C lapi get t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi get t fields are listed in Table 13 in the order that they occur in the lapi xfer t structure.

Table 13. LAPI_Get and lapi_get_t equivalents

lapi_get_t field name (C)	lapi_get_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Get parameter
Xfer_type	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C
			LAPI_Xfer value in FORTRAN: LAPI_GET_XFER
flags	int	INTEGER(KIND = 4)	none
			LAPI_Xfer parameter in FORTRAN: flags
tgt	uint	INTEGER(KIND = 4)	tgt
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN: <i>pad</i>
tgt_addr	lapi_long_t	INTEGER(KIND = 8)	tgt_addr
org_addr	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_addr

Table 13. LAPI_Get and lapi_get_t equivalents (continued)

lapi_get_t field name (C)	lapi_get_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Get parameter
len	ulong	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	len
tgt_cntr	lapi_long_t	INTEGER(KIND = 8)	tgt_cntr
org_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_cntr
chndlr	compl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: chndlr
cinfo	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: cinfo

When the origin data buffer has completely arrived, the pointer to the completion handler (chndlr) is called with the completion data (cinfo), if chndlr is not a NULL pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI_Get.

lapi_getv_t details

Table 14 shows the correspondence among the parameters of the LAPI_Getv subroutine, the fields of the C lapi_getv_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_getv_t fields are listed in Table 13 on page 568 in the order that they occur in the lapi_xfer_t structure.

Table 14. LAPI_Getv and lapi_getv_t equivalents

lapi_getv_t field name (C)	lapi_getv_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Getv parameter
Xfer_type	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C
			LAPI_Xfer value in FORTRAN: LAPI_GETV_XFER
flags	int	INTEGER(KIND = 4)	none
			LAPI_Xfer parameter in FORTRAN: <i>flags</i>
tgt	uint	INTEGER(KIND = 4)	tgt
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (64-bit): pad
org_vec	lapi_vec_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_vec
tgt_vec	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	tgt_vec
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (32-bit): pad
tgt_cntr	lapi_long_t	INTEGER(KIND = 8)	tgt_cntr
org_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_cntr
chndlr	compl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none
			LAPI_Xfer parameter in FORTRAN: chndlr

Table 14. LAPI_Getv and lapi_getv_t equivalents (continued)

lapi_getv_t field name (C)	lapi_getv_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Getv parameter
cinfo	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>cinfo</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (32-bit): pad2

For this release, the flags field accepts USE_TGT_VEC_TYPE (see "The lapi_amdgsp_t flags field" on page 567) to indicate that tgt_vec is to be interpreted as type lapi_vec_t; otherwise, it is interpreted as type lapi_lvec_t. Note the corresponding field is lapi_vec_t in the LAPI_Getv call.

When the origin data buffer has completely arrived, the pointer to the completion handler (chndlr) is called with the completion data (cinfo) if chndlr is not a NULL pointer (in C) or LAPI ADDR NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI Getv.

lapi_put_t details

Table 15 shows the correspondence among the parameters of the LAPI_Put subroutine, the fields of the C lapi_put_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_put_t fields are listed in Table 15 in the order that they occur in the lapi_xfer_t structure.

Table 15. LAPI_Put and lapi_put_t equivalents

lapi_put_t field name (C)	lapi_put_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Put parameter
Xfer_type	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C LAPI_Xfer value in FORTRAN: LAPI_PUT_XFER
flags	int	INTEGER(KIND = 4)	none LAPI_Xfer parameter in FORTRAN: flags
tgt	uint	INTEGER(KIND = 4)	tgt
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN: <i>pad</i>
tgt_addr	lapi_long_t	INTEGER(KIND = 8)	tgt_addr
org_addr	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_addr
len	ulong	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	len
shdlr	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: shdlr
sinfo	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>sinfo</i>
tgt_cntr	lapi_long_t	INTEGER(KIND = 8)	tgt_cntr
org_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_cntr

Table 15. LAPI_Put and lapi_put_t equivalents (continued)

lapi_put_t field name (C)	lapi_put_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Put parameter
cmpl_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	cmpl_cntr

When the origin data buffer is free to be used, the pointer to the send completion handler (shdlr) is called with the send completion data (sinfo), if shdlr is not a NULL pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI Put.

lapi_putv_t details

Table 16 shows the correspondence among the parameters of the LAPI_Putv subroutine, the fields of the C lapi_putv_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_putv_t fields are listed in Table 15 on page 570 in the order that they occur in the lapi xfer t structure.

Table 16. LAPI_Putv and lapi_putv_t equivalents

lapi_putv_t field name (C)	lapi_putv_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Putv parameter
Xfer_type	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C
			LAPI_Xfer value in FORTRAN: LAPI_PUT_XFER
flags	int	INTEGER(KIND = 4)	none
			LAPI_Xfer parameter in FORTRAN: flags
tgt	uint	INTEGER(KIND = 4)	tgt
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (64-bit): pad
shdlr	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit)	none
		INTEGER(KIND = 8) (64-bit)	LAPI_Xfer parameter in FORTRAN: shdlr
sinfo	void *	INTEGER(KIND = 4) (32-bit)	none
		INTEGER(KIND = 8) (64-bit)	LAPI_Xfer parameter in FORTRAN: sinfo
org_vec	lapi_vec_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_vec
tgt_vec	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	tgt_vec
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (32-bit): pad
tgt_cntr	lapi_long_t	INTEGER(KIND = 8)	tgt_cntr
org_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_cntr
cmpl_cntr	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	cmpl_cntr

For this release, the flags field accepts USE_TGT_VEC_TYPE (see "The lapi_amdgsp_t flags field" on page 567) to indicate that tgt vec is to be interpreted as lapi vec t; otherwise, it is interpreted as a lapi lvec t. Note that the corresponding field is lapi vec t in the LAPI Putv call.

When the origin data buffer is free to be modified, the pointer to the send completion handler (shdlr) is called with the send completion data (sinfo), if shdlr is not a NULL pointer (in C) or LAPI ADDR NULL (in FORTRAN). Otherwise, the behavior is identical to that of LAPI_Putv.

lapi_rmw_t details

Table 17 shows the correspondence among the parameters of the LAPI_Rmw subroutine, the fields of the C lapi rmw t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi rmw t fields are listed in Table 15 on page 570 in the order that they occur in the lapi xfer t structure.

Table 17. LAPI_Rmw and lapi_rmw_t equivalents

lapi_rmw_t field name (C)	lapi_rmw_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Rmw parameter
Xfer_type	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C LAPI_Xfer value in FORTRAN: LAPI_RMW_XFER
ор	Rmw_ops_t	INTEGER(KIND = 4)	ор
tgt	uint	INTEGER(KIND = 4)	tgt
size	uint	INTEGER(KIND = 4)	implicit in C
			LAPI_Xfer parameter in FORTRAN: size (must be 32 or 64)
tgt_var	lapi_long_t	INTEGER(KIND = 8)	tgt_var
in_val	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	in_val
prev_tgt_val	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	prev_tgt_val
org_cntr	lapi_cntr t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	org_cntr
shdlr	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit)	none
		INTEGER(KIND = 8) (64-bit)	LAPI_Xfer parameter in FORTRAN: shdlr
sinfo	void *	INTEGER(KIND = 4) (32-bit)	none
		INTEGER(KIND = 8) (64-bit)	LAPI_Xfer parameter in FORTRAN: shdlr
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (32-bit): pad

When the origin data buffer is free to be used, the pointer to the send completion handler (shdlr) is called with the send completion data (sinfo), if shdlr is not a NULL pointer (in C) or LAPI_ADDR_NULL (in FORTRAN). The size value must be either 32 or 64, indicating whether you want the in val and prev tat val fields to point to a 32-bit or 64-bit quantity, respectively. Otherwise, the behavior is identical to that of LAPI Rmw.

Parameters

INPUT

hndl Specifies the LAPI handle.

xfer cmd Specifies the name and parameters of the data transfer function.

OUTPUT

Restrictions

Interoperability between 32-bit and 64-bit tasks using the LAPI_GETV_XFER and LAPI_PUTV_XFER transfer types in **LAPI_Xfer** is not supported in this release.

C Examples

1. To run the sample code shown in **LAPI Get** using the **LAPI Xfer** interface:

```
lapi xfer t xfer struct;
/* initialize the table buffer for the data addrsesses */
/* get remote data buffer addresses */
LAPI Address init(hndl,(void *)data buffer,data buffer list);
/* retrieve data len bytes from address data buffer list[tgt] on */
/* task tgt. write the data starting at address data buffer.
/* tgt_cntr and org_cntr can be NULL.
xfer_struct.Get.Xfer_type = LAPI_GET_XFER;
xfer struct.Get.flags = 0;
xfer struct.Get.tgt = tgt;
xfer_struct.Get.tgt_addr = data_buffer_list[tgt];
xfer_struct.Get.org_addr = data_buffer;
xfer struct.Get.len = data len;
xfer_struct.Get.tgt_cntr = tgt_cntr;
xfer_struct.Get.org_cntr = org_cntr;
LAPI Xfer(hndl, &xfer struct);
```

2. To implement the LAPI_STRIDED_VECTOR example from LAPI_Amsendv using the LAPI_Xfer interface:

```
lapi xfer t xfer struct;
                                                /* info for LAPI Xfer call */
lapi vec t
                                               /* data for data transfer */
            vec;
                               /* NUM VECS = number of vectors to transfer */
vec->num vecs = NUM VECS;
                               /* must match that of the target vector */
vec->vec type = LAPI GEN STRIDED XFER;
                                                 /* same as target vector */
vec->info[0] = buffer address; /* starting address for data copy
                                                                          */
vec->info[1] = block_size;
                                /* bytes of data to copy
vec->info[2] = stride;
                                /* distance from copy block to copy block */
/* data will be copied as follows:
                                                                          */
/* block size bytes will be copied from buffer address
                                                                          */
/* block_size bytes will be copied from buffer_address+stride
                                                                          */
/* block_size bytes will be copied from buffer_address+(2*stride)
                                                                          */
/* block size bytes will be copied from buffer address+(3*stride)
/* block size bytes will be copied from buffer address+((NUM VECS-1)*stride) */
```

```
xfer struct.Amv.Xfer type = LAPI AMV XFER;
xfer struct.Amv.flags
                         = 0;
xfer struct.Amv.tgt
                         = tgt;
xfer struct.Amv.hdr hdl = hdr hdl list[tgt];
xfer struct.Amv.uhdr len = uhdr len; /* user header length */
xfer struct.Amv.uhdr
                         = uhdr;
/* LAPI AMV XFER allows the use of a send completion handler */
/* If non-null, the shdlr function is invoked at the point
/* the origin counter would increment. Note that both the
/* org cntr and shdlr can be used.
/* The user's shdlr must be of type scompl_hndlr_t *.
                                                             */
/* scompl hndlr t is defined in /usr/include/lapi.h
xfer struct.shdlr = shdlr;
/* Use sinfo to pass user-defined data into the send
                                                            */
/* completion handler, if desired.
xfer struct.sinfo
                     = sinfo;
                                    /* send completion data */
xfer struct.org vec = vec;
xfer struct.tgt cntr = tgt cntr;
xfer_struct.org_cntr = org_cntr;
xfer struct.cmpl cntr = cmpl cntr;
LAPI Xfer(hndl, &xfer struct);
```

See LAPI_Amsendv for more information about the header handler definition.

Return Values

}

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_DATA_LEN Indicates that the value of *udata_len* or *len* is greater than the value of

LAPI constant LAPI MAX MSG SZ.

LAPI ERR DGSP Indicates that the DGSP that was passed in is NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN) or is not a registered DGSP.

LAPI_ERR_DGSP_ATOM Indicates that the DGSP has an atom_size that is less than 0 or greater

than MAX_ATOM_SIZE.

LAPI_ERR_DGSP_BRANCH Indicates that the DGSP attempted a branch that fell outside the code

array.

LAPI_ERR_DGSP_CTL Indicates that a DGSP control instruction was encountered in a non-valid

context (such as a gather-side control or scatter-side control with an atom

size of **0** at gather, for example).

LAPI ERR DGSP OPCODE Indicates that the DGSP op-code is not valid.

Indicates that the DGSP has greater GOSUB depth than the allocated LAPI_ERR_DGSP_STACK

stack supports. Stack allocation is specified by the dgsp->depth member.

LAPI ERR HDR HNDLR NULL

Indicates that the hdr hdl passed in is NULL (in C) or LAPI ADDR NULL

(in FORTRAN).

LAPI ERR HNDL INVALID Indicates that the *hndl* passed in is not valid (not initialized or in

terminated state).

LAPI_ERR_IN_VAL_NULL Indicates that the in val pointer is NULL (in C) or LAPI ADDR NULL (in

FORTRAN).

LAPI_ERR_MEMORY_EXHAUSTED

LAPI is unable to obtain memory from the system.

LAPI ERR OP SZ Indicates that the **lapi rmw t** size field is not set to **32** or **64**.

LAPI_ERR_ORG_ADDR_NULL

Indicates either that the udata parameter passed in is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN) and *udata_len* is greater than **0**, or that the org_addr passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) and *len* is greater than **0**.

Note: if *Xfer_type* = **LAPI_DGSP_XFER**, the case in which *udata* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) and udata_len is greater than **0** is valid, so an error is not returned.

LAPI ERR ORG EXTENT Indicates that the org vec's extent (stride * num vecs) is greater than the

value of LAPI constant LAPI MAX MSG SZ.

LAPI ERR ORG STRIDE Indicates that the *org vec* stride is less than block.

LAPI_ERR_ORG_VEC_ADDR

Indicates that the org vec->info[i] is NULL (in C) or LAPI ADDR NULL (in

FORTRAN), but its length (org vec->len[i]) is not **0**.

LAPI_ERR_ORG_VEC_LEN Indicates that the sum of org vec->len is greater than the value of LAPI

constant LAPI MAX MSG SZ.

LAPI_ERR_ORG_VEC_NULL Indicates that the org_vec value is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN).

LAPI_ERR_ORG_VEC_TYPE Indicates that the *org_vec->vec_type* is not valid.

LAPI ERR RMW OP Indicates the op is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL

Indicates that the strided vector address org_vec->info[0] is NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL

Indicates that the strided vector address tgt_vec->info[0] is NULL (in C) or

LAPI_ADDR_NULL (in FORTRAN).

LAPI ERR TGT Indicates that the tgt passed in is outside the range of tasks defined in the

job.

LAPI_ERR_TGT_ADDR_NULL

Indicates that ret_addr is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN).

LAPI ERR TGT EXTENT Indicates that tgt vec's extent (stride * num vecs) is greater than the

value of LAPI constant LAPI MAX MSG SZ.

LAPI ERR TGT PURGED Indicates that the subroutine returned early because LAPI Purge totask()

was called.

LAPI ERR TGT STRIDE Indicates that the tat vec stride is less than block.

LAPI ERR TGT VAR NULL Indicates that the *tgt_var* address is NULL (in C) or that the value of

tgt var is LAPI ADDR NULL (in FORTRAN).

LAPI_ERR_TGT_VEC_ADDR Indicates that the tgt_vec->info[i] is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN), but its length (tgt vec->len[i]) is not **0**.

Indicates that the sum of tgt vec->len is greater than the value of LAPI LAPI_ERR_TGT_VEC_LEN

constant LAPI MAX MSG SZ.

LAPI_ERR_TGT_VEC_NULL Indicates that tgt_vec is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN).

LAPI ERR TGT VEC TYPE Indicates that the tat vec->vec type is not valid.

LAPI_ERR_UHDR_LEN Indicates that the *uhdr_len* value passed in is greater than

MAX_UHDR_SZ or is not a multiple of the processor's doubleword size.

LAPI_ERR_UHDR_NULL Indicates that the *uhdr* passed in is NULL (in C) or LAPI_ADDR_NULL (in

FORTRAN), but uhdr_len is not 0.

LAPI_ERR_VEC_LEN_DIFF Indicates that *org_vec* and *tgt_vec* have different lengths (*len[]*).

LAPI_ERR_VEC_NUM_DIFF Indicates that *org_vec* and *tgt_vec* have different *num_vecs*.

LAPI_ERR_VEC_TYPE_DIFF

Indicates that org vec and tgt vec have different vector types (vec type).

LAPI ERR XFER CMD Indicates that the Xfer cmd is not valid.

Location

/usr/lib/liblapi_r.a

Related Information

Books: RSCT for AIX 5L: LAPI Programming Guide for information about bulk message transfer

Subroutines: LAPI_Amsend, LAPI_Amsendv, LAPI_Get, LAPI_Getv, LAPI_Putv, LAPI_Rmw

layout object create Subroutine

Purpose

Initializes a layout context.

Library

Layout Library (libi18n.a)

Syntax

#include <sys/lc layout.h>

int layout_object_create (locale_name, layout_object) const char * locale name; LayoutObject * layout object;

Description

The layout_object_create subroutine creates the LayoutObject structure associated with the locale specified by the *locale name* parameter. The **LayoutObject** structure is a symbolic link containing all the data and methods necessary to perform the layout operations on context dependent and bidirectional characters of the locale specified.

When the layout object create subroutine completes without errors, the layout object parameter points to a valid LayoutObject structure that can be used by other BIDI subroutines. The returned LayoutObject structure is initialized to an initial state that defines the behavior of the BIDI subroutines. This initial state is locale dependent and is described by the layout values returned by the layout object getvalue

subroutine. You can change the layout values of the LayoutObject structure using the layout object setvalue subroutine. Any state maintained by the LayoutObject structure is independent of the current global locale set with the setlocale subroutine.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs

Parameters

Specifies a locale. It is recommended that you use the LC_CTYPE category by calling the locale name

setlocale (LC_CTYPE, NULL) subroutine.

Points to a valid LayoutObject structure that can be used by other layout subroutines. This layout_object

parameter is used only when the layout object create subroutine completes without

errors.

The layout object parameter is not set and a non-zero value is returned if a valid

LayoutObject structure cannot be created.

Return Values

Upon successful completion, the layout_object_create subroutine returns a value of 0. The layout_object parameter points to a valid handle.

Error Codes

If the **layout object create** subroutine fails, it returns the following error codes:

LAYOUT_EINVAL The locale specified by the *locale_name* parameter is not available.

LAYOUT EMFILE The OPEN MAX value of files descriptors are currently open in the calling process.

LAYOUT_ENOMEM Insufficient storage space is available.

Related Information

The "layout object editshape or wcslayout object editshape Subroutine," "layout object free Subroutine" on page 588, "layout object getvalue Subroutine" on page 580, "layout object setvalue Subroutine" on page 582, "layout object shapeboxchars Subroutine" on page 584, "layout object transform or wcslayout object transform Subroutine" on page 585.

Bidirectionality and Character Shaping and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

layout_object_editshape or wcslayout_object_editshape Subroutine

Purpose

Edits the shape of the context text.

Library

Layout library (libi18n.a)

Syntax

```
#include <sys/lc_layout.h>
int layout editshape ( layout object, EditType, index, InpBuf, Inpsize, OutBuf, OutSize)
LayoutObject layout object;
BooleanValue EditType;
size_t *index;
const char *InpBuf;
size t *Inpsize;
void *OutBuf;
size t *OutSize;
int wcslayout object editshape(layout object, EditType, index, InpBuf, Inpsize, OutBuf, OutSize)
LayoutObject layout_object;
BooleanValue EditType;
size t *index;
const wchar t *InpBuf;
size_t *InpSize;
void *OutBuf;
size t *OutSize;
```

Description

The <code>layout_object_editshape</code> and <code>wcslayout_object_editshape</code> subroutines provide the shapes of the context text. The shapes are defined by the code element specified by the <code>index</code> parameter and any surrounding code elements specified by the ShapeContextSize layout value of the <code>LayoutObject</code> structure. The <code>layout_object</code> parameter specifies this <code>LayoutObject</code> structure.

Use the **layout_object_editshape** subroutine when editing code elements of one byte. Use the **wcslayout_object_editshape** subroutine when editing single code elements of multibytes. These subroutines do not affect any state maintained by the **layout_object_transform** or **wcslayout_object_transform** subroutine.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs

Parameters

layout_object

Specifies the LayoutObject structure created by the layout_object_create subroutine.

EditType

Specifies the type of edit shaping. When the EditType parameter stipulates the EditInput field, the subroutine reads the current code element defined by the index parameter and any preceding code elements defined by ShapeContextSize layout value of the LayoutObject structure. When the EditType parameter stipulates the EditReplace field, the subroutine reads the current code element defined by the index parameter and any surrounding code elements defined by ShapeContextSize layout value of the LayoutObject structure.

Note: The editing direction defined by the Orientation and TEXT_VISUAL of the TypeOfText layout values of the LayoutObject structure determines which code elements are preceding and succeeding.

When the ActiveShapeEditing layout value of the LayoutObject structure is set to True, the LayoutObject structure maintains the state of the EditInput field that may affect subsequent calls to these subroutines with the EditInput field defined by the EditType parameter. The state of the EditInput field of LayoutObject structure is not affected when the EditType parameter is set to the EditReplace field. To reset the state of the EditInput field to its initial state, call these subroutines with the *InpBuf* parameter set to NULL. The state of the EditInput field is not affected if errors occur within the subroutines. Specifies an offset (in bytes) to the start of a code element in the InpBuf parameter on input. The InpBuf parameter provides the base text to be edited. In addition, the context of the surrounding code elements is considered where the minimum set of code elements needed for the specific context dependent script(s) is identified by the ShapeContextSize layout value.

If the set of surrounding code elements defined by the index, InpBuf, and InpSize parameters is less than the size of front and back of the ShapeContextSize layout value, these subroutines assume there is no additional context available. The caller must provide the minimum context if it is available. The *index* parameter is in units associated with the type of the InpBuf parameter.

On return, the index parameter is modified to indicate the offset to the first code element of the InpBuf parameter that required shaping. The number of code elements that required shaping is indicated on return by the InpSize parameter.

Specifies the source to be processed. A Null value with the EditInput field in the EditType parameter indicates a request to reset the state of the EditInput field to its initial state.

Any portion of the *InpBuf* parameter indicates the necessity for redrawing or shaping. Specifies the number of code elements to be processed in units on input. These units are associated with the types for these subroutines. A value of -1 indicates that the input is delimited by a Null code element.

On return, the value is modified to the actual number of code elements needed by the InpBuf parameter. A value of 0 when the value of the EditType parameter is the EditInput field indicates that the state of the EditInput field is reset to its initial state. If the OutBuf parameter is not NULL, the respective shaped code elements are written into the OutBuf parameter.

Contains the shaped output text. You can specify this parameter as a Null pointer to indicate that no transformed text is required. If Null, the subroutines return the index and InpSize parameters, which specify the amount of text required, to be redrawn.

The encoding of the OutBuf parameter depends on the ShapeCharset layout value defined in layout object parameter. If the ActiveShapeEditing layout value is set to False, the encoding of the OutBuf parameter is to be the same as the code set of the locale associated with the specified LayoutObject structure.

index

InpBuf

InpSize

OutBuf

OutSize

Specifies the size of the output buffer on input in number of bytes. Only the code elements required to be shaped are written into the OutBuf parameter.

The output buffer should be large enough to contain the shaped result; otherwise, only partial shaping is performed. If the ActiveShapeEditing layout value is set to True, the OutBuf parameter should be allocated to contain at least the number of code elements in the InpBuf parameter multiplied by the value of the ShapeCharsetSize layout value.

On return, the OutSize parameter is modified to the actual number of bytes placed in the output buffer.

When the OutSize parameter is specified as 0, the subroutines calculate the size of an output buffer large enough to contain the transformed text from the input buffer. The result will be returned in this field. The content of the buffers specifies by the InpBuf and OutBuf parameters, and the value of the *InpSize* parameter, remain unchanged.

Return Values

Upon successful completion, these subroutines return a value of 0. The index and InpSize parameters return the minimum set of code elements required to be redrawn.

Error Codes

If these subroutines fail, they return the following error codes:

LAYOUT EILSEQ Shaping stopped due to an input code element that cannot be shaped. The

> index parameter indicates the code element that caused the error. This code element is either a valid code element that cannot be shaped according to the ShapeCharset layout value or an invalid code element not defined by the code set defined in the LayoutObject structure. Use the mbtowc or wctomb subroutine in the same locale as the LayoutObject structure to determine if

the code element is valid.

LAYOUT_E2BIG The output buffer is too small and the source text was not processed. The

index and InpSize parameters are not guaranteed on return.

LAYOUT_EINVAL Shaping stopped due to an incomplete code element or shift sequence at the

end of input buffer. The InpSize parameter indicates the number of code

elements successfully transformed.

Note: You can use this error code to determine the code element causing the

error.

LAYOUT ERANGE Either the *index* parameter is outside the range as defined by the *InpSize*

parameter, more than 15 embedding levels are in the source text, or the InpBuf parameter contains unbalanced Directional Format (Push/Pop).

Related Information

The "layout_object_create Subroutine" on page 576, "layout_object_free Subroutine" on page 588, "layout_object_getvalue Subroutine," "layout_object_setvalue Subroutine" on page 582, "layout object shapeboxchars Subroutine" on page 584, "layout object transform or wcslayout object transform Subroutine" on page 585.

Bidirectionality and Character Shaping and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

layout object getvalue Subroutine

Purpose

Queries the current layout values of a **LayoutObject** structure.

Library

Layout Library (libi18n.a)

Syntax

```
#include <sys/lc layout.h>
```

```
int layout object getvalue( layout object, values, index)
LayoutObject layout object;
LayoutValues values;
int *index;
```

Description

The layout_object_getvalue subroutine queries the current setting of layout values within the LayoutObject structure. The layout_object parameter specifies the LayoutObject structure created by the layout object create subroutine.

The name field of the Layout Values structure contains the name of the layout value to be queried. The value field is a pointer to where the layout value is stored. The values are queried from the LayoutObject structure and represent its current state.

For example, if the layout value to be queried is of type T, the *value* parameter must be of type T*. If T itself is a pointer, the layout object getvalue subroutine allocates space to store the actual data. The caller must free this data by calling the **free(T)** subroutine with the returned pointer.

When setting the value field, an extra level of indirection is present that is not present using the layout object setvalue parameter. When you set a layout value of type T, the value field contains T. However, when guerying the same layout value, the value field contains &T.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs

Parameters

layout_object Specifies the LayoutObject structure created by the layout_object_create subroutine. values

Specifies an array of layout values of type Layout ValueRec that are to be queried in the

LayoutObject structure. The end of the array is indicated by name=0.

Specifies a layout value to be queried. If the value cannot be gueried, the index parameter index

causing the error is returned and the subroutine returns a non-zero value.

Return Values

Upon successful completion, the layout_object_getvalue subroutine returns a value of 0. All layout values were successfully queried.

Error Codes

If the **layout object getvalue** subroutine fails, it returns the following values:

LAYOUT_EINVAL The layout value specified by the index parameter is unknown or the layout object

parameter is invalid.

LAYOUT EMOMEM Insufficient storage space is available.

Examples

The following example queries whether the locale is bidirectional and gets the values of the in and out orienations.

```
#include <sys/lc layout.h>
#include <locale.h>
main()
LayoutObject plh;
int RC=0;
LayoutValues layout;
LayoutTextDescriptor Descr;
int index;
RC=layout object create(setlocale(LC CTYPE, ""), &plh); /* create object */
if (RC) {printf("Create error !!\n"); exit(0);}
layout=malloc(3*sizeof(LayoutValueRec));
                                         /* allocate layout array */
layout[0].name=ActiveBidirection;
                                         /* set name */
layout[1].name=Orientation;
                                         /* set name */
layout[1].value=(caddr t)&Descr;
           /* send address of memory to be allocated by function */
layout[2].name=0;
                                      /* indicate end of array */
RC=layout object getvalue(plh,layout,&index);
if (RC) {printf("Getvalue error at %d !!\n",index); exit(0);}
printf("ActiveBidirection = %d \n",*(layout[0].value));
printf("Orientation in = %x out = %x \n", Descr->>in, Descr->>out);
free(layout);
                                     /* free layout array */
free (Descr);
                          /* free memory allocated by function */
RC=layout object free(plh);
                                    /* free layout object */
if (RC) printf("Free error !!\n");
```

Related Information

The "layout_object_create Subroutine" on page 576, "layout_object_editshape or wcslayout_object_editshape Subroutine" on page 577, "layout_object_free Subroutine" on page 588, "layout_object_setvalue Subroutine," "layout_object_shapeboxchars Subroutine" on page 584, and "layout_object_transform or wcslayout_object_transform Subroutine" on page 585.

Bidirectionality and Character Shaping and National Language Support Overview in *AIX 5L Version 5.2 National Language Support Guide and Reference*.

layout_object_setvalue Subroutine

Purpose

Sets the layout values of a LayoutObject structure.

Library

Layout Library (libi18n.a)

Syntax

#include <sys/lc layout.h>

```
int layout_object_setvalue( layout object, values, index)
LayoutObject layout object;
LayoutValues values;
int *index;
```

Description

The layout_object_setvalue subroutine changes the current layout values of the LayoutObject structure. The layout_object parameter specifies the LayoutObject structure created by the layout_object_create subroutine. The values are written into the LayoutObject structure and may affect the behavior of subsequent layout functions.

Note: Some layout values do alter internal states maintained by a LayoutObject structure.

The name field of the LayoutValueRec structure contains the name of the layout value to be set. The value field contains the actual value to be set. The value field is large enough to support all types of layout values. For more information on layout value types, see "Layout Values for the Layout Library" in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs

Parameters

layout_object Specifies the LayoutObject structure returned by the layout_object_create subroutine.

values Specifies an array of layout values of the type LayoutValueRec that this subroutine sets. The end

of the array is indicated by name=0.

Specifies a layout value to be queried. If the value cannot be gueried, the index parameter index

causing the error is returned and the subroutine returns a non-zero value. If an error is

generated, a subset of the values may have been previously set.

Return Values

Upon successful completion, the layout_object_setvalue subroutine returns a value of 0. All layout values were successfully set.

Error Codes

If the **layout object setvalue** subroutine fails, it returns the following values:

LAYOUT_EINVAL The layout value specified by the index parameter is unknown, its value is invalid, or the

layout_object parameter is invalid.

LAYOUT EMFILE The (OPEN_MAX) file descriptors are currently open in the calling process.

LAYOUT ENOMEM Insufficient storage space is available.

Examples

The following example sets the TypeofText value to Implicit and the out value to Visual.

```
#include <sys/lc layout.h>
#include <locale.h>
main()
LayoutObject plh;
int RC=0;
LavoutValues lavout:
LayoutTextDescriptor Descr;
```

```
int index:
RC=layout object create(setlocale(LC CTYPE,""),&plh); /* create object */
if (RC) {printf("Create error !!\n"); exit(0);}
layout=malloc(2*sizeof(LayoutValueRec)); /*allocate layout array*/
Descr=malloc(sizeof(LayoutTextDescriptorRec)); /* allocate text descriptor */
layout[0].name=TypeOfText;
                                   /* set name */
layout[0].value=(caddr_t)Descr;
                                     /* set value */
layout[1].name=0;
                                     /* indicate end of array */
Descr->in=TEXT IMPLICIT;
Descr->out=TEXT VISUAL; RC=layout object setvalue(plh,layout,&index);
if (RC) printf("SetValue error at %d!!\n",index); /* check return code */
                                   /* free allocated memory */
free(layout);
free (Descr);
RC=layout object free(plh);
                                   /* free layout object */
if (RC) printf("Free error !!\n");
```

Related Information

The "layout_object_create Subroutine" on page 576, "layout_object_editshape or wcslayout_object_editshape Subroutine" on page 577, "layout_object_free Subroutine" on page 588, "layout_object_getvalue Subroutine" on page 580, "layout_object_shapeboxchars Subroutine," and "layout_object_transform or wcslayout_object_transform Subroutine" on page 585.

Bidirectionality and Character Shaping and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

layout_object_shapeboxchars Subroutine

Purpose

Shapes box characters.

Library

Layout Library (libi18n.a)

Syntax

```
#include <sys/lc_layout.h>
int layout_object_shapeboxchars( layout_object, InpBuf, InpSize, OutBuf)
LayoutObject layout_object;
const char *InpBuf;
const size_t InpSize;
char *OutBuf;
```

Description

The layout_object_shapeboxchars subroutine shapes box characters into the VT100 box character set.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*

Parameters

layout_object Specifies the LayoutObject structure created by the layout_object_create subroutine.

InpBuf Specifies the source text to be processed.

InpSize Specifies the number of code elements to be processed.

OutBuf Contains the shaped output text.

Return Values

Upon successful completion, this subroutine returns a value of 0.

Error Codes

If this subroutine fails, it returns the following values:

LAYOUT_EILSEQ Shaping stopped due to an input code element that cannot be mapped into the VT100 box

character set.

LAYOUT EINVAL Shaping stopped due to an incomplete code element or shift sequence at the end of the

input buffer.

Related Information

The "layout_object_create Subroutine" on page 576, "layout_object_editshape or wcslayout_object_editshape Subroutine" on page 577, "layout_object_free Subroutine" on page 588, "layout_object_getvalue Subroutine" on page 580, "layout_object_setvalue Subroutine" on page 582, and "layout object transform or wcslayout object transform Subroutine."

Bidirectionality and Character Shaping and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

layout object transform or wcslayout object transform Subroutine

Purpose

Transforms text according to the current layout values of a LayoutObject structure.

Library

Layout Library (libi18n.a)

Syntax

```
#include <sys/lc_layout.h>
int layout object transform ( layout object, InpBuf, InpSize, OutBuf, OutSize, InpToOut, OutToInp, BidiLvl)
LayoutObject layout object;
const char *InpBuf;
size t *InpSize;
void * OutBuf;
size t *OutSize;
size t *InpToOut;
size_t *OutToInp;
unsigned char *BidiLvl;
int wcslayout_object_transform (layout_object, InpBuf, InpSize, OutBuf, OutSize, InpToOut, OutToInp, BidiLvl)
LayoutObject layout object;
const char *InpBuf:
size_t *InpSize;
void *OutBuf;
```

Size t *OutSize; size_t *InpToOut; size t *OutToInp; unsigned char *BidiLvl;

Description

The layout_object_transform and wcslayout_object_transform subroutines transform the text specified by the InpBuf parameter according to the current layout values in the LayoutObject structure. Any layout value whose type is LayoutTextDescriptor describes the attributes within the InpBuf and OutBuf parameters. If the attributes are the same as the InpBuf and OutBuf parameters themselves, a null transformation is done with respect to that specific layout value.

The output of these subroutines may be one or more of the following results depending on the setting of the respective parameters:

OutBuf, OutSize Any transformed data is stored in the *OutBuf* parameter.

InpToOut A cross reference from each code element of the InpBuf parameter to the transformed

OutToInp A cross reference to each code element of the InpBuf parameter from the transformed

BidiLvI A weighted value that represents the directional level of each code element of the

InpBuf parameter. The level is dependent on the internal directional algorithm of the

LayoutObject structure.

You can specify each of these output parameters as Null to indicate that no output is needed for the specific parameter. However, you should set at least one of these parameters to a nonNULL value to perform any significant work.

To perform shaping of a text string without reordering of code elements, set the TypeOfText layout value to TEXT_VISUAL and the in and out values of the Orientation layout value alike. These layout values are in the LayoutObject structure.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs

Parameters

layout_object Specifies the LayoutObject structure created by the layout_object_create subroutine.

InpBuf Specifies the source text to be processed. This parameter cannot be null.

Specifies the units of code elements processed associated with the bytes for the InpSize

layout object transform and wcslayout object transform subroutines. A value of -1 indicates that the input is delimited by a null code element. On return, the value is modified to the actual number of code elements processed in the InBuf parameter. However, if the value in the OutSize parameter is zero, the value of the InpSize parameter is not changed.

Contains the transformed data. You can specify this parameter as a null pointer to indicate

that no transformed data is required.

The encoding of the OutBuf parameter depends on the ShapeCharset layout value defined in the LayoutObject structure. If the ActiveShapeEditing layout value is set to True, the encoding of the OutBuf parameter is the same as the code set of the locale associated with

the LayoutObject structure.

OutBuf

OutSize

InpToOut

OutToInp

BidiLvl

Specifies the size of the output buffer in number of bytes. The output buffer should be large enough to contain the transformed result; otherwise, only a partial transformation is performed. If the ActiveShapeEditing layout value is set to True, the OutBuf parameter should be allocated to contain at least the number of code elements multiplied by the ShapeCharsetSize layout value.

On return, the OutSize parameter is modified to the actual number of bytes placed in this parameter.

When you specify the OutSize parameter as 0, the subroutine calculates the size of an output buffer to be large enough to contain the transformed text. The result is returned in this field. The content of the buffers specified by the InpBuf and OutBuf parameters, and a value of the *InpSize* parameter remains unchanged.

Represents an array of values with the same number of code elements as the InpBuf parameter if InpToOut parameter is not a null pointer.

On output, the nth value in InpToOut parameter corresponds to the nth code element in InpBuf parameter. This value is the index in OutBuf parameter which identifies the transformed ShapeCharset element of the nth code element in InpBuf parameter. You can specify the InpToOut parameter as null if no index array from the InpBuf to OutBuf

parameters is desired.

Represents an array of values with the same number of code elements as contained in the

OutBuf parameter if the OutToInp parameter is not a null pointer.

On output, the nth value in the OutToInp parameter corresponds to the nth ShapeCharset element in the OutBuf parameter. This value is the index in the InpBuf parameter which identifies the original code element of the nth ShapeCharset element in the OutBuf parameter. You can specify the OutToInp parameter as NULL if no index array from the

OutBuf to InpBuf parameters is desired.

Represents an array of values with the same number of elements as the source text if the BidiLvI parameter is not a null pointer. The nth value in the BidiLvI parameter corresponds

to the nth code element in the InpBuf parameter. This value is the level of this code element as determined by the bidirectional algorithm. You can specify the BidiLvI parameter

as null if a levels array is not desired.

Return Values

Upon successful completion, these subroutines return a value of 0.

Error Codes

LAYOUT_EILSEQ

If these subroutines fail, they return the following values:

Transformation stopped due to an input code element that cannot be shaped or is invalid. The InpSize parameter indicates the number of the

code element successfully transformed.

Note: You can use this error code to determine the code element

causing the error.

This code element is either a valid code element but cannot be shaped into the ShapeCharset layout value or is an invalid code element not defined by the code set of the locale of the LayoutObject structure. You can use the **mbtowc** and **wctomb** subroutines to determine if the code element is valid when used in the same locale as the LayoutObject

structure.

The output buffer is full and the source text is not entirely processed.

LAYOUT_E2BIG

LAYOUT_EINVAL Transformation stopped due to an incomplete code element or shift

sequence at the end of the input buffer. The InpSize parameter indicates

the number of the code elements successfully transformed.

Note: You can use this error code to determine the code element

causing the error.

LAYOUT_ERANGE More than 15 embedding levels are in the source text or the InpBuf

parameter contains unbalanced Directional Format (Push/Pop).

When the size of OutBuf parameter is not large enough to contain the

entire transformed text, the input text state at the end of the

LAYOUT_E2BIG error code is returned. To resume the transformation on the remaining text, the application calls the layout_object_transform subroutine with the same LayoutObject structure, the same InpBuf

parameter, and InpSize parameter set to 0.

Examples

The following is an example of transformation of both directional re-ordering and shaping.

Notes:

- 1. Uppercase represent left-to-right characters; lowercase represent right-to-left characters.
- 2. xyz represent the shapes of cde.

Position: 0123456789 AB cde 12Z InpBuf: 0123456789 Position: OutBuf: AB 12 zyxZ 0123456789 0128765349 Position: ToTarget: Position: 0123456789 ToSource: 0127865439 rusicion: BidiLevel: 0123456789 0001111220

Related Information

The "layout_object_create Subroutine" on page 576, "layout_object_editshape or wcslayout_object_editshape Subroutine" on page 577, "layout_object_free Subroutine," "layout_object_getvalue Subroutine" on page 580, "layout_object_setvalue Subroutine" on page 582, and "layout object shapeboxchars Subroutine" on page 584.

Bidirectionality and Character Shaping and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

layout_object_free Subroutine

Purpose

Frees a LayoutObject structure.

Library

Layout library (libi18n.a)

Syntax

#include <sys/lc_layout.h>

```
int layout_object_free(layout object)
LayoutObject layout object;
```

Description

The layout_object_free subroutine releases all the resources of the LayoutObject structure created by the **layout object create** subroutine. The *layout object* parameter specifies this **LayoutObject** structure.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs

Parameters

layout_object

Specifies a LayoutObject structure returned by the layout_object_create subroutine.

Return Values

Upon successful completion, the layout object free subroutine returns a value of 0. All resources associated with the layout object parameter are successfully deallocated.

Error Codes

If the **layout object free** subroutine fails, it returns the following error code:

LAYOUT_EFAULT

Errors occurred while processing the request.

Related Information

The "layout object create Subroutine" on page 576, "layout object editshape or wcslayout_object_editshape Subroutine" on page 577, "layout_object_getvalue Subroutine" on page 580, "layout_object_setvalue Subroutine" on page 582, "layout_object_shapeboxchars Subroutine" on page 584, and "layout_object_transform or wcslayout_object_transform Subroutine" on page 585.

Bidirectionality and Character Shaping and National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

Idahread Subroutine

Purpose

Reads the archive header of a member of an archive file.

Library

Object File Access Routine Library (libld.a)

Syntax

```
#include <stdio.h>
#include <ar.h>
#include <1dfcn.h>
int ldahread( ldPointer, ArchiveHeader)
LDFILE *ldPointer;
ARCHDR *ArchiveHeader;
```

Description

If the **TYPE**(*IdPointer*) macro from the **Idfcn.h** file is the archive file magic number, the **Idahread** subroutine reads the archive header of the extended common object file currently associated with the *IdPointer* parameter into the area of memory beginning at the *ArchiveHeader* parameter.

Parameters

IdPointer Points to the LDFILE structure that was returned as the result of a successful call to

Idopen or Idaopen.

ArchiveHeader Points to a **ARCHDR** structure.

Return Values

The Idahread subroutine returns a SUCCESS or FAILURE value.

Error Codes

The **Idahread** routine fails if the **TYPE**(*IdPointer*) macro does not represent an archive file, or if it cannot read the archive header.

Related Information

The **Idfhread** ("Idfhread Subroutine" on page 592) subroutine, **Idgetname** ("Idgetname Subroutine" on page 594) subroutine, **Idlread**, **Idlinit**, or **Idlitem** ("Idlread, Idlinit, or Idlitem Subroutine" on page 596) subroutine, **Idshread** or **Idnshread** ("Idshread or Idnshread Subroutine" on page 602) subroutine, **Idtbread** ("Idtbread Subroutine" on page 606) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idclose or Idaclose Subroutine

Purpose

Closes a common object file.

Library

Object File Access Routine Library (libld.a)

Syntax

#include <stdio.h>
#include <ldfcn.h>

int ldclose(ldPointer)
LDFILE *ldPointer;

int ldaclose(ldPointer)
LDFILE *ldPointer;

Description

The **Idopen** and **Idclose** subroutines provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of common object files can be processed as if it were a series of simple common object files.

If the Idfcn.h file TYPE(IdPointer) macro is the magic number of an archive file, and if there are any more files in the archive, the Idclose subroutine reinitializes the Idfcn.h file OFFSET(IdPointer) macro to the file address of the next archive member and returns a failure value. The Idfile structure is prepared for a subsequent Idopen.

If the TYPE(IdPointer) macro does not represent an archive file, the Idclose subroutine closes the file and frees the memory allocated to the Idfile structure associated with IdPointer.

The Idaclose subroutine closes the file and frees the memory allocated to the Idfile structure associated with the IdPointer parameter regardless of the value of the TYPE(IdPointer) macro.

Parameters

IdPointer

Pointer to the LDFILE structure that was returned as the result of a successful call to Idopen or Idaopen.

Return Values

The **Idclose** subroutine returns a SUCCESS or FAILURE value.

The Idaclose subroutine always returns a SUCCESS value and is often used in conjunction with the Idaopen subroutine.

Error Codes

The **Idclose** subroutine returns a failure value if there are more files to archive.

Related Information

The Idaopen or Idopen ("Idopen or Idaopen Subroutine" on page 599) subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idexp, Idexpf, or Idexpl Subroutine

Purpose

Loads exponent of a floating-point number.

Syntax

```
#include <math.h>
float ldexpf(x, exp)
float x;
int exp;
long double ldexpl (x, exp)
long double x;
int exp;
double ldexp (x, exp)
double x;
int exp;
```

Description

The **Idexpf**, **Idexpl**, and **Idexp** subroutines compute the quantity $x * 2^{exp}$.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE ALL EXCEPT) before calling these functions. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed. X

Specifies the exponent of 2. exp

Return Values

Upon successful completion, the **Idexpf**, **Idexpf**, and **Idexp** subroutines return x multiplied by 2, raised to the power exp.

If the **Idexpf**, **Idexpl**, or **Idexp** subroutines would cause overflow, a range error occurs and the **Idexpf**, Idexpl, and Idexp subroutines return ±HUGE VALF, ±HUGE VALL, and ±HUGE VAL (according to the sign of x), respectively.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 is returned.

If x is NaN, a NaN is returned.

If x is ± 0 or Inf, x is returned.

If exp is 0, x is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

Error Codes

If the result of the Idexp or Idexpl subroutine overflows, then +/- HUGE VAL is returned, and the global variable errno is set to ERANGE.

If the result of the **Idexp** or **Idexpl** subroutine underflows, 0 is returned, and the **errno** global variable is set to a **ERANGE** value.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, class, finite, isnan, or unordered Subroutines" on page 138

math.h in AIX 5L Version 5.2 Files Reference.

Idfhread Subroutine

Purpose

Reads the file header of an XCOFF file.

Library

Object File Access Routine Library (libld.a)

Syntax

```
#include <stdio.h>
#include <1dfcn.h>
int ldfhread ( ldPointer, FileHeader)
LDFILE *ldPointer;
void *FileHeader;
```

Description

The **Idfhread** subroutine reads the file header of the object file currently associated with the *IdPointer* parameter into the area of memory beginning at the FileHeader parameter. For AIX 4.3.2 and above, it is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the file header of the associated object file. Since the Idopen subroutine provides magic number information (via the **HEADER**(IdPointer).f_magic macro), the calling application can always determine whether the FileHeader pointer should refer to a 32-bit FILHDR or 64-bit FILHDR 64 structure.

Parameters

IdPointer Points to the LDFILE structure that was returned as the result of a successful call to Idopen or

Idaopen subroutine.

FileHeader Points to a buffer large enough to accommodate a FILHDR structure, according to the object

mode of the file being read.

Return Values

The **Idfhread** subroutine returns Success or Failure.

Error Codes

The **Idfhread** subroutine fails if it cannot read the file header.

Note: In most cases, the use of Idfhread can be avoided by using the HEADER (IdPointer) macro defined in the Idfcn.h file. The information in any field or fieldname of the header file may be accessed using the header (IdPointer) fieldname macro.

Examples

The following is an example of code that opens an object file, determines its mode, and uses the Idfhread subroutine to acquire the file header. This code would be compiled with both _XCOFF32_ and _XCOFF64_ defined:

```
#define __XCOFF32__
#define __XCOFF64__
#include <ldfcn.h>
/* for each FileName to be processed */
if ( (ldPointer = ldopen(fileName, ldPointer)) != NULL)
             FileHead32;
    FILHDR
    FILHDR 64 FileHead64;
    void
              *FileHeader:
    if ( HEADER(1dPointer).f_magic == U802TOCMAGIC )
        FileHeader = &FileHead32;
    else if ( HEADER(ldPointer).f magic == U803XTOCMAGIC )
        FileHeader = &FileHead64;
    else
```

```
FileHeader = NULL;
    if (FileHeader && (ldfhread( ldPointer, &FileHeader ) == SUCCESS) )
        /* ...successfully read header... */
       /* ...process according to magic number... */
}
```

Related Information

The Idahread ("Idahread Subroutine" on page 589) subroutine, Idgetname ("Idgetname Subroutine") subroutine, Idlread, Idlinit, or Idlitem ("Idlread, Idlinit, or Idlitem Subroutine" on page 596) subroutine, Idopen ("Idopen or Idaopen Subroutine" on page 599) subroutine, Idshread or Idnshread ("Idshread or Idnshread Subroutine" on page 602) subroutine, Idtbread ("Idtbread Subroutine" on page 606) subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idgetname Subroutine

Purpose

Retrieves symbol name for common object file symbol table entry.

Library

Object File Access Routine Library (libld.a)

Syntax

```
#include <stdio.h>
#include <1dfcn.h>
char *ldgetname ( ldPointer, Symbol)
LDFILE *ldPointer;
void *Symbol;
```

Description

The **Idgetname** subroutine returns a pointer to the name associated with *Symbol* as a string. The string is in a static buffer local to the **Idgetname** subroutine that is overwritten by each call to the **Idgetname** subroutine and must therefore be copied by the caller if the name is to be saved.

The common object file format handles arbitrary length symbol names with the addition of a string table. The **Idgetname** subroutine returns the symbol name associated with a symbol table entry for an XCOFF-format object file.

The calling routine to provide a pointer to a buffer large enough to contain a symbol table entry for the associated object file. Since the Idopen subroutine provides magic number information (via the **HEADER**(IdPointer).f magic macro), the calling application can always determine whether the Symbol pointer should refer to a 32-bit SYMENT or 64-bit SYMENT_64 structure.

The maximum length of a symbol name is **BUFSIZ**, defined in the **stdio.h** file.

Parameters

IdPointer Points to an LDFILE structure that was returned as the result of a successful call to the Idopen or

Idaopen subroutine.

Symbol Points to an initialized 32-bit or 64-bit **SYMENT** structure.

Error Codes

The **Idgetname** subroutine returns a null value (defined in the **stdio.h** file) for a COFF-format object file if the name cannot be retrieved. This situation can occur if one of the following is true:

- · The string table cannot be found.
- The string table appears invalid (for example, if an auxiliary entry is handed to the Idgetname subroutine wherein the name offset lies outside the boundaries of the string table).
- The name's offset into the string table is past the end of the string table.

Typically, the **Idgetname** subroutine is called immediately after a successful call to the **Idtbread** subroutine to retrieve the name associated with the symbol table entry filled by the **Idtbread** subroutine.

Examples

The following is an example of code that determines the object file type before making a call to the **Idtbread** and **Idgetname** subroutines.

```
#define __XCOFF32__
#define __XCOFF64__
#include <ldfcn.h>
          Symbol32;
SYMENT
SYMENT_64 Symbol64;
void
         *Symbol;
if ( HEADER(1dPointer).f magic == U802TOCMAGIC )
    Symbol = \&Symbol32;
else if ( HEADER(1dPointer).f magic == U64 TOCMAGIC )
    Symbol = &Symbol64;
else
    Symbol = NULL;
if (Symbol)
    /* for each symbol in the symbol table */
    for ( symnum = 0 ; symnum < HEADER(ldPointer).f nsyms ; symnum++ )</pre>
        if ( ldtbread(ldPointer.symnum.Symbol) == SUCCESS )
            char *name = ldgetname(ldPointer,Symbol)
            if ( name )
                 /* Got the name... */
            }
            /* Increment symnum by the number of auxiliary entries */
            if ( HEADER(1dPointer).f_magic == U802TOCMAGIC )
                 symnum += Symbol32.n_numaux;
            else if ( HEADER(1dPointer).f magic == U64 TOCMAGIC )
                 symnum += Symbol64.n numaux;
        else
```

Related Information

The **Idahread** ("Idahread Subroutine" on page 589) subroutine, **Idfhread** ("Idfhread Subroutine" on page 592) subroutine, **Idlied**, **Idlinit**, or **Idlied** ("Idlied, Idlinit, or Idlied Subroutine") subroutine, **Idshread** or **Idnshread** ("Idshread or Idnshread Subroutine" on page 602) subroutine, **Idthread** ("Idthread Subroutine" on page 606) subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idlread, Idlinit, or Idlitem Subroutine

Purpose

Manipulates line number entries of a common object file function.

Library

Object File Access Routine Library (libld.a)

Syntax

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

int ldlread ( ldPointer, FunctionIndex, LineNumber, LineEntry)
LDFILE *ldPointer;
int FunctionIndex;
unsigned short LineNumber;
void *LineEntry;

int ldlinit (ldPointer, FunctionIndex)
LDFILE *ldPointer;
int FunctionIndex;

int ldlitem (ldPointer, LineNumber, LineEntry)
LDFILE *ldPointer;
unsigned short LineNumber;
void *LineEntry;
```

Description

The **Idlread** subroutine searches the line number entries of the XCOFF file currently associated with the *IdPointer* parameter. The **Idlread** subroutine begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by the *FunctionIndex* parameter, the index of its entry in the object file symbol table. The **Idlread** subroutine reads the entry with the smallest line number equal to or greater than the *LineNumber* parameter into the memory beginning at the *LineEntry* parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the line number entry for the associated object file type. Since the **Idopen** subroutine provides magic number information (via the **HEADER**(*IdPointer*).**f_magic** macro), the calling application can always determine whether the *LineEntry* pointer should refer to a 32-bit LINENO or 64-bit LINENO_64 structure.

The **Idlinit** and **Idlitem** subroutines together perform the same function as the **Idlread** subroutine. After an initial call to the Idlread or Idlinit subroutine, the Idlitem subroutine may be used to retrieve successive line number entries associated with a single function. The Idlinit subroutine simply locates the line number entries for the function identified by the FunctionIndex parameter. The Idlitem subroutine finds and reads the entry with the smallest line number equal to or greater than the LineNumber parameter into the memory beginning at the *LineEntry* parameter.

Parameters

Points to the LDFILE structure that was returned as the result of a successful call to the **IdPointer**

Idopen, or Idaopen subroutine.

LineNumber Specifies the index of the first *LineNumber* parameter entry to be read.

Points to a buffer that will be filled in with a LINENO structure from the object file. LineEntry

FunctionIndex Points to the symbol table index of a function.

Return Values

The Idlread, Idlinit, and Idlitem subroutines return a SUCCESS or FAILURE value.

Error Codes

The **Idlread** subroutine fails if there are no line number entries in the object file, if the *FunctionIndex* parameter does not index a function entry in the symbol table, or if it finds no line number equal to or greater than the LineNumber parameter. The Idlinit subroutine fails if there are no line number entries in the object file or if the FunctionIndex parameter does not index a function entry in the symbol table. The **Idlitem** subroutine fails if it finds no line number equal to or greater than the *LineNumber* parameter.

Related Information

The Idahread ("Idahread Subroutine" on page 589) subroutine, Idfhread ("Idfhread Subroutine" on page 592) subroutine, **Idgetname** ("Idgetname Subroutine" on page 594) subroutine, **Idshread** or **Idnshread** ("Idshread or Idnshread Subroutine" on page 602) subroutine, Idtbread ("Idtbread Subroutine" on page 606) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idlseek or IdnIseek Subroutine

Purpose

Seeks to line number entries of a section of a common object file.

Library

Object File Access Routine Library (libld.a)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>
int ldlseek ( ldPointer, SectionIndex)
LDFILE *ldPointer;
unsigned short SectionIndex;
```

int ldnlseek (ldPointer, SectionName)
LDFILE *ldPointer;
char *SectionName;

Description

The **Idlseek** subroutine seeks to the line number entries of the section specified by the *SectionIndex* parameter of the common object file currently associated with the *IdPointer* parameter. The first section has an index of 1.

The **IdnIseek** subroutine seeks to the line number entries of the section specified by the *SectionName* parameter.

Both subroutines determine the object mode of the associated file before seeking to the relocation entries of the indicated section.

Parameters

IdPointer Points to the LDFILE structure that was returned as the result of a successful call to the

Idopen or Idaopen subroutine.

SectionIndex Specifies the index of the section whose line number entries are to be seeked to.

SectionName Specifies the name of the section whose line number entries are to be seeked to.

Return Values

The Idlseek and IdnIseek subroutines return a SUCCESS or FAILURE value.

Error Codes

The **Idlseek** subroutine fails if the *SectionIndex* parameter is greater than the number of sections in the object file. The **IdnIseek** subroutine fails if there is no section name corresponding with the *SectionName* parameter. Either function fails if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Related Information

The **Idohseek** ("Idohseek Subroutine") subroutine, **Idrseek** or **Idnrseek** ("Idrseek or Idnrseek Subroutine" on page 601) subroutine, **Idsseek** or **Idnsseek** ("Idsseek or Idnsseek Subroutine" on page 604) subroutine, **Idtbseek** ("Idtbseek Subroutine" on page 607) subroutine.

Subroutines, Example Programs, and Libraries in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.*

Idohseek Subroutine

Purpose

Seeks to the optional file header of a common object file.

Library

Object File Access Routine Library (libld.a)

Syntax

#include <stdio.h>
#include <ldfcn.h>

```
int ldohseek ( ldPointer)
LDFILE *ldPointer;
```

Description

The Idohseek subroutine seeks to the optional auxiliary header of the common object file currently associated with the IdPointer parameter. The subroutine determines the object mode of the associated file before seeking to the end of its file header.

Parameters

IdPointer

Points to the LDFILE structure that was returned as the result of a successful call to Idopen or Idaopen subroutine.

Return Values

The Idohseek subroutine returns a SUCCESS or FAILURE value.

Error Codes

The Idohseek subroutine fails if the object file has no optional header, if the file is not a 32-bit or 64-bit object file, or if it cannot seek to the optional header.

Related Information

The Idlseek or IdnIseek ("Idlseek or IdnIseek Subroutine" on page 597) subroutine, Idrseek or Idnrseek ("Idrseek or Idnrseek Subroutine" on page 601) subroutine, Idsseek or Idnsseek ("Idsseek or Idnsseek Subroutine" on page 604) subroutine, Idtbseek ("Idtbseek Subroutine" on page 607) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idopen or Idaopen Subroutine

Purpose

Opens an object or archive file for reading.

Library

Object File Access Routine Library (libld.a)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>
LDFILE *ldopen( FileName, ldPointer)
char *FileName:
LDFILE *ldPointer;
LDFILE *Idaopen(FileName, ldPointer)
char *FileName;
LDFILE *ldPointer;
LDFILE *lddopen(FileDescriptor, type, ldPointer)
```

int FileDescriptor; char *type; **LDFILE** *ldPointer;

Description

The **Idopen** and **Idclose** subroutines provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of object files can be processed as if it were a series of ordinary object files.

If the *IdPointer* is null, the **Idopen** subroutine opens the file named by the *FileName* parameter and allocates and initializes an LDFILE structure, and returns a pointer to the structure.

If the IdPointer parameter is not null and refers to an LDFILE for an archive, the structure is updated for reading the next archive member. In this case, and if the value of the TYPE(IdPointer) macro is the archive magic number **ARTYPE**.

The **Idopen** and **Idclose** subroutines are designed to work in concert. The **Idclose** subroutine returns failure only when the IdPointer refers to an archive containing additional members. Only then should the **Idopen** subroutine be called with a num-null *IdPointer* argument. In all other cases, in particular whenever a new FileName parameter is opened, the Idopen subroutine should be called with a null IdPointer argument.

If the value of the *IdPointer* parameter is not null, the **Idaopen** subroutine opens the *FileName* parameter again and allocates and initializes a new LDFILE structure, copying the TYPE, OFFSET, and HEADER fields from the *IdPointer* parameter. The **Idaopen** subroutine returns a pointer to the new **Idfile** structure. This new pointer is independent of the old pointer, *IdPointer*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

The **Iddopen** function accesses the previously opened file referenced by the *FileDescriptor* parameter. In all other respects, it functions the same as the Idopen subroutine.

For AIX 4.3.2 and above, the functions transparently open both 32-bit and 64-bit object files, as well as both small format and large format archive files. Once a file or archive is successfully opened, the calling application can examine the HEADER(IdPointer).f_magic field to check the magic number of the file or archive member associated with IdPointer. (This is necessary due to an archive potentially containing members that are not object files.) The magic numbers U802TOCMAGIC and (for AIX 4.3.2 and above) U803XTOCMAGIC are defined in the **Idfcn.h** file. If the value of **TYPE(IdPointer)** is the archive magic number ARTYPE, the flags field can be checked for the archive type. Large format archives will have the flag bit AR_TYPE_BIG set in LDFLAGS(IdPointer). Large format archives are available on AIX 4.3 and later.

Parameters

FileName Specifies the file name of an object file or archive.

IdPointer Points to an LDFILE structure. FileDescriptor Specifies a valid open file descriptor.

Points to a character string specifying the mode for the open file. The fdopen function is type

used to open the file.

Error Codes

Both the **Idopen** and **Idopen** subroutines open the file named by the *FileName* parameter for reading. Both functions return a null value if the FileName parameter cannot be opened, or if memory for the LDFILE structure cannot be allocated.

A successful open does not ensure that the given file is a common object file or an archived object file.

Examples

The following is an example of code that uses the **Idopen** and **Idclose** subroutines:

```
/* for each FileName to be processed */
ldPointer = NULL:
if((ldPointer = ldopen(FileName, ldPointer)) != NULL)
                      /* check magic number */
                      /* process the file */
  while(ldclose(ldPointer) == FAILURE );
```

Related Information

The Idclose or Idaclose ("Idclose or Idaclose Subroutine" on page 590) subroutine, fopen, fopen64, freopen, freopen64, or fdopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idrseek or Idnrseek Subroutine

Purpose

Seeks to the relocation entries of a section of an XCOFF file.

Library

Object File Access Routine Library (libld.a)

Syntax

```
#include <stdio.h>
#include <1dfcn.h>
int ldrseek ( ldPointer, SectionIndex)
ldfile *ldPointer;
unsigned short SectionIndex;
int ldnrseek (ldPointer, SectionName)
ldfile *ldPointer;
char *SectionName;
```

Description

The Idrseek subroutine seeks to the relocation entries of the section specified by the SectionIndex parameter of the common object file currently associated with the *IdPointer* parameter.

The Idnrseek subroutine seeks to the relocation entries of the section specified by the SectionName parameter.

For AIX 4.3.2 and above, both subroutines determine the object mode of the associated file before seeking to the relocation entries of the indicated section.

Parameters

IdPointer Points to an LDFILE structure that was returned as the result of a successful call to the

Idopen, Iddopen, or Idaopen subroutines.

SectionIndex Specifies an index for the section whose relocation entries are to be sought. SectionName Specifies the name of the section whose relocation entries are to be sought.

Return Values

The Idrseek and Idnrseek subroutines return a SUCCESS or FAILURE value.

Error Codes

The Idrseek subroutine fails if the contents of the SectionIndex parameter are greater than the number of sections in the object file. The **Idnrseek** subroutine fails if there is no section name corresponding with the SectionName parameter. Either function fails if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note: The first section has an index of 1.

Related Information

The Idohseek ("Idohseek Subroutine" on page 598) subroutine, Idlseek or IdnIseek ("Idlseek or IdnIseek Subroutine" on page 597) subroutine, Idsseek or Idnsseek ("Idsseek or Idnsseek Subroutine" on page 604) subroutine, **Idtbseek** ("Idtbseek Subroutine" on page 607) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idshread or Idnshread Subroutine

Purpose

Reads a section header of an XCOFF file.

Library

Object File Access Routine Library (libld.a)

Syntax

```
#include <stdio.h>
#include <1dfcn.h>
int ldshread ( ldPointer, SectionIndex, SectionHead)
LDFILE *ldPointer;
unsigned short SectionIndex;
void *SectionHead;
int ldnshread (ldPointer, SectionName, SectionHead)
LDFILE *ldPointer;
char *SectionName:
void *SectionHead;
```

Description

The **Idshread** subroutine reads the section header specified by the *SectionIndex* parameter of the common object file currently associated with the *IdPointer* parameter into the area of memory beginning at the location specified by the *SectionHead* parameter.

The **Idnshread** subroutine reads the section header named by the *SectionName* argument into the area of memory beginning at the location specified by the *SectionHead* parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the section header of the associated object file. Since the **Idopen** subroutine provides magic number information (via the **HEADER**(*IdPointer*).f_magic macro), the calling application can always determine whether the *SectionHead* pointer should refer to a 32-bit **SCNHDR** or 64-bit **SCNHDR** 64 structure.

Only the first section header named by the *SectionName* argument is returned by the **Idshread** subroutine.

Parameters

IdPointer Points to an LDFILE structure that was returned as the result of a successful call to the

Idopen, Ildopen, or Idaopen subroutine.

SectionIndex Specifies the index of the section header to be read.

Note: The first section has an index of 1.

SectionHead Points to a buffer large enough to accept either a 32-bit or a 64-bit SCNHDR structure,

according to the object mode of the file being read.

Specifies the name of the section header to be read.

Return Values

The Idshread and Idnshread subroutines return a SUCCESS or FAILURE value.

Error Codes

The **Idshread** subroutine fails if the *SectionIndex* parameter is greater than the number of sections in the object file. The **Idnshread** subroutine fails if there is no section with the name specified by the *SectionName* parameter. Either function fails if it cannot read the specified section header.

Examples

The following is an example of code that opens an object file, determines its mode, and uses the **Idnshread** subroutine to acquire the .text section header. This code would be compiled with both **XCOFF32** and **XCOFF64** defined:

```
else
    SectionHeader = NULL;

if ( SectionHeader && (ldnshread( ldPointer, ".text", &SectionHeader ) == SUCCESS) )
{
    /* ...successfully read header... */
    /* ...process according to magic number... */
}
```

Related Information

The **Idahread** ("Idahread Subroutine" on page 589) subroutine, **Idfhread** ("Idfhread Subroutine" on page 592) subroutine, **Idgetname** ("Idgetname Subroutine" on page 594) subroutine, **Idlread**, **Idlinit**, or **Idlitem** ("Idlread, Idlinit, or Idlitem Subroutine" on page 596) subroutine, **Idtbread** ("Idtbread Subroutine" on page 606) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idsseek or Idnsseek Subroutine

Purpose

Seeks to an indexed or named section of a common object file.

Library

Object File Access Routine Library (libld.a)

Syntax

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

int ldsseek ( ldPointer, SectionIndex)
LDFILE *ldPointer;
unsigned short SectionIndex;

int ldnsseek (ldPointer, SectionName)
LDFILE *ldPointer;
char *SectionName;
```

Description

The **Idsseek** subroutine seeks to the section specified by the *SectionIndex* parameter of the common object file currently associated with the *IdPointer* parameter. The subroutine determines the object mode of the associated file before seeking to the indicated section.

The **Idnsseek** subroutine seeks to the section specified by the *SectionName* parameter.

Parameters

IdPointer Points to the LDFILE structure that was returned as the result of a successful call to the

Idopen or Idaopen subroutine.

SectionIndex Specifies the index of the section whose line number entries are to be seeked to.

SectionName Specifies the name of the section whose line number entries are to be seeked to.

The Idsseek and Idnsseek subroutines return a SUCCESS or FAILURE value.

Error Codes

The **Idsseek** subroutine fails if the SectionIndex parameter is greater than the number of sections in the object file. The Idnsseek subroutine fails if there is no section name corresponding with the SectionName parameter. Either function fails if there is no section data for the specified section or if it cannot seek to the specified section.

Note: The first section has an index of 1.

Related Information

The Idlseek or Idnlseek ("Idlseek or Idnlseek Subroutine" on page 597) subroutine, Idohseek ("Idohseek Subroutine" on page 598) subroutine, Idrseek or Idnrseek ("Idrseek or Idnrseek Subroutine" on page 601) subroutine, Idtbseek ("Idtbseek Subroutine" on page 607) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idtbindex Subroutine

Purpose

Computes the index of a symbol table entry of a common object file.

Library

Object File Access Routine Library (libld.a)

Syntax

#include <stdio.h> #include <1dfcn.h>

long ldtbindex (ldPointer) **LDFILE** **ldPointer*;

Description

The **Idtbindex** subroutine returns the index of the symbol table entry at the current position of the common object file associated with the IdPointer parameter.

The index returned by the **Idtbindex** subroutine may be used in subsequent calls to the **Idtbread** subroutine. However, since the **Idtbindex** subroutine returns the index of the symbol table entry that begins at the current position of the object file, if the Idtbindex subroutine is called immediately after a particular symbol table entry has been read, it returns the index of the next entry.

Parameters

IdPointer

Points to the LDFILE structure that was returned as a result of a successful call to the Idopen or Idaopen subroutine.

The Idtbindex subroutine returns the value BADINDEX upon failure. Otherwise a value greater than or equal to zero is returned.

Error Codes

The Idtbindex subroutine fails if there are no symbols in the object file or if the object file is not positioned at the beginning of a symbol table entry.

Note: The first symbol in the symbol table has an index of 0.

Related Information

The Idtbread ("Idtbread Subroutine") subroutine, Idtbseek ("Idtbseek Subroutine" on page 607) subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idtbread Subroutine

Purpose

Reads an indexed symbol table entry of a common object file.

Library

Object File Access Routine Library (libld.a)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>
int ldtbread ( ldPointer, SymbolIndex, Symbol)
LDFILE *ldPointer;
long SymbolIndex;
void *Symbol;
```

Description

The Idtbread subroutine reads the symbol table entry specified by the SymbolIndex parameter of the common object file currently associated with the *IdPointer* parameter into the area of memory beginning at the Symbol parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the symbol table entry of the associated object file. Since the Idopen subroutine provides magic number information (via the **HEADER**(IdPointer).f magic macro), the calling application can always determine whether the Symbol pointer should refer to a 32-bit SYMENT or 64-bit SYMENT 64 structure.

Parameters

IdPointer Points to the LDFILE structure that was returned as the result of a successful call to the

Idopen or Idaopen subroutine.

SymbolIndex Specifies the index of the symbol table entry to be read. Symbol Points to a either a 32-bit or a 64-bit **SYMENT** structure.

The Idtbread subroutine returns a SUCCESS or FAILURE value.

Error Codes

The **Idtbread** subroutine fails if the *SymbolIndex* parameter is greater than or equal to the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note: The first symbol in the symbol table has an index of 0.

Related Information

The Idahread ("Idahread Subroutine" on page 589) subroutine, Idfhread ("Idfhread Subroutine" on page 592) subroutine, Idgetname ("Idgetname Subroutine" on page 594) subroutine, Idlread, Idlinit, or Idlitem ("Idlread, Idlinit, or Idlitem Subroutine" on page 596) subroutine, Idshread or Idnshread ("Idshread or Idnshread Subroutine" on page 602) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Idtbseek Subroutine

Purpose

Seeks to the symbol table of a common object file.

Library

Object File Access Routine Library (libld.a)

Syntax

#include <stdio.h> #include <ldfcn.h>

int ldtbseek (ldPointer) **LDFILE** *ldPointer;

Description

The Idtbseek subroutine seeks to the symbol table of the common object file currently associated with the *IdPointer* parameter.

Parameters

IdPointer

Points to the LDFILE structure that was returned as the result of a successful call to the Idopen or Idaopen subroutine.

Return Values

The Idtbseek subroutine returns a SUCCESS or FAILURE value.

Error Codes

The **Idtbseek** subroutine fails if the symbol table has been stripped from the object file or if the subroutine cannot seek to the symbol table.

Related Information

The Idlseek or Idnlseek ("Idlseek or Idnlseek Subroutine" on page 597) subroutine, Idohseek ("Idohseek Subroutine" on page 598) subroutine, Idrseek or Idnrseek ("Idrseek or Idnrseek Subroutine" on page 601) subroutine, Idsseek or Idnsseek ("Idsseek or Idnsseek Subroutine" on page 604) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Igamma, Igammaf, or Igammal Subroutine

Purpose

Computes the log gamma.

Syntax

```
#include <math.h>
extern int signgam;
double lgamma(x)
double x;
float lgammaf (x)
float x;
long double lgammal (x)
long double x;
```

Description

The sign of $\Gamma(x)$ is returned in the external integer **signgam**.

The **Igamma**, **Igammaf**, and **Igammal** subroutines are not reentrant. A function that is not required to be reentrant is not required to be thread-safe.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE ALL EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the Igamma, Igammaf, and Igammal subroutines return the logarithmic gamma of x.

If x is a non-positive integer, a pole error shall occur and **Igamma**, **Igammaf**, and **Igammal** will return +HUGE VAL, +HUGE VALF, and +HUGE VALL.

If the correct value would cause overflow, a range error shall occur and Igamma, Igammaf, and Igammal will return ±HUGE VAL, ±HUGE VALF, ±HUGE VALL, respectively.

If x is NaN, a NaN is returned.

If x is 1 or 2, +0 is returned.

If x is $\pm \ln f$, $+\ln f$ is returned.

Related Information

"exp, expf, or expl Subroutine" on page 205, "feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, _class, finite, isnan, or unordered Subroutines" on page 138.

math.h in AIX 5L Version 5.2 Files Reference.

lineout Subroutine

Purpose

Formats a print line.

Library

None (provided by the print formatter)

Syntax

#include <piostruct.h>

int lineout (fileptr) FILE *fileptr;

Description

The **lineout** subroutine is invoked by the formatter driver only if the **setup** subroutine returns a non-null pointer. This subroutine is invoked for each line of the document being formatted. The lineout subroutine reads the input data stream from the *fileptr* parameter. It then formats and outputs the print line until it recognizes a situation that causes vertical movement on the page.

The lineout subroutine should process all characters to be printed and all printer commands related to horizontal movement on the page.

The lineout subroutine should not output any printer commands that cause vertical movement on the page. Instead, it should update the vpos (new vertical position) variable pointed to by the shars vars structure that it shares with the formatter driver to indicate the new vertical position on the page. It should also refresh the shar vars variables for vertical increment and vertical decrement (reverse line-feed) commands.

When the lineout subroutine returns, the formatter driver sends the necessary commands to the printer to advance to the new vertical position on the page. This position is specified by the vpos variable. The formatter driver automatically handles top and bottom margins, new pages, initial pages to be skipped, and progress reports to the qdaemon daemon.

The following conditions can cause vertical movements:

- · Line-feed control character or variable line-feed control sequence
- · Vertical-tab control character
- Form-feed control character
- · Reverse line-feed control character
- · A line too long for the printer that wraps to the next line

Other conditions unique to a specific printer also cause vertical movement.

Parameters

fileptr

Specifies a file structure for the input data stream.

Return Values

Upon successful completion, the **lineout** subroutine returns the number of bytes processed from the input data stream. It excludes the end-of-file character and any control characters or escape sequences that result only in vertical movement on the page (for example, line feed or vertical tab).

If a value of 0 is returned and the value in the **vpos** variable pointed to by the **shars_vars** structure has not changed, or there are no more data bytes in the input data stream, the formatter driver assumes that printing is complete.

If the **lineout** subroutine detects an error, it uses the **piomsgout** subroutine to issue an error message. It then invokes the **pioexit** subroutine with a value of PIOEXITBAD.

Note: If either the **piocmdout** or **piogetstr** subroutine detects an error, it automatically issues its own error messages and terminates the print job.

Related Information

The **piocmdout** subroutine, **pioexit** subroutine, **piogetstr** subroutine, **piomsgout** subroutine, **setup** subroutine.

Adding a New Printer Type to Your System and Printer Addition Management Subsystem: Programming Overview in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

Example of Print Formatter in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

link Subroutine

Purpose

Creates an additional directory entry for an existing file.

Library

Standard C Library (libc.a)

Syntax

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

```
int link ( Path1, Path2)
const char *Path1, *Path2;
```

Description

The **link** subroutine creates an additional hard link (directory entry) for an existing file. Both the old and the new links share equal access rights to the underlying object.

Parameters

Path1 Points to the path name of an existing file.

Path2 Points to the path name of the directory entry to be created.

Notes:

- 1. If Network File System (NFS) is installed on your system, these paths can cross into another node.
- 2. With hard links, both the Path1 and Path2 parameters must reside on the same file system. If Path1 is a symbolic link, an error is returned. Creating links to directories requires root user authority.

Return Values

Upon successful completion, the link subroutine returns a value of 0. Otherwise, a value of -1 is returned, and the errno global variable is set to indicate the error.

Error Codes

The **link** subroutine is unsuccessful if one of the following is true:

EACCES EDQUOT	Indicates the requested link requires writing in a directory that denies write permission. Indicates the directory in which the entry for the new link is being placed cannot be extended, or disk blocks could not be allocated for the link because the user or group quota of disk blocks or i-nodes on the file system containing the directory has been exhausted.
EEXIST	Indicates the link named by the Path2 parameter already exists.
EMLINK	Indicates the file already has the maximum number of links.
ENOENT	Indicates the file named by the Path1 parameter does not exist.
ENOSPC	Indicates the directory in which the entry for the new link is being placed cannot be extended because there is no space left on the file system containing the directory.
EPERM	Indicates the file named by the <i>Path1</i> parameter is a directory, and the calling process does not have root user authority.
EROFS	Indicates the requested link requires writing in a directory on a read-only file system.
EXDEV	Indicates the link named by the Path2 parameter and the file named by the Path1 parameter are on

The link subroutine can be unsuccessful for other reasons. See Appendix A, "Base Operating System Error Codes for Services That Require Path-Name Resolution," on page 1001 for a list of additional errors.

If NFS is installed on the system, the link subroutine is unsuccessful if the following is true:

different file systems, or the file named by Path1 refers to a named STREAM.

ETIMEDOUT Indicates the connection timed out.

Related Information

The **symlink** subroutine, **unlink** subroutine.

The **link** or **unlink** command, **In** command, **rm** command.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

lio_listio or lio_listio64 Subroutine

The lio_listio or lio_listio64 subroutine includes information for the POSIX AIO lio_listio subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO lio_listio subroutine.

POSIX AIO lio_listio Subroutine

Purpose

Initiates a list of asynchronous I/O requests with a single call.

Syntax

```
#include <aio.h>
int lio listio(mode, list, nent, sig)
int mode;
struct aiocb *restrict const list[restrict];
int nent;
struct sigevent *restrict sig;
```

Description

The lio listio subroutine initiates a list of I/O requests with a single function call.

The mode parameter takes one of the values (LIO WAIT, LIO NOWAIT or LIO NOWAIT AIOWAIT) declared in <aio.h> and determines whether the subroutine returns when the I/O operations have been completed, or as soon as the operations have been queued. If the *mode* parameter is set to LIO WAIT. the subroutine waits until all I/O is complete and the sig parameter is ignored.

If the mode parameter is set to LIO_NOWAIT or LIO_NOWAIT_AIOWAIT, the subroutine returns immediately. If LIO NOWAIT is set, asynchronous notification occurs, according to the sig parameter, when all I/O operations complete. If sig is NULL, no asynchronous notification occurs. If sig is not NULL, asynchronous notification occurs when all the requests in list have completed. If LIO NOWAIT AIOWAIT is set, the aio_nwait subroutine must be called for the aio control blocks to be updated. For more information, see the "aio_nwait Subroutine" on page 29.

The I/O requests enumerated by *list* are submitted in an unspecified order.

The list parameter is an array of pointers to aiocb structures. The array contains nent elements. The array may contain NULL elements, which are ignored.

The aio_lio_opcode field of each aiocb structure specifies the operation to be performed. The supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in <aio.h>. The LIO NOP operation causes the list entry to be ignored. If the aio lio opcode element is equal to LIO_READ, an I/O operation is submitted as if by a call to aio_read with the aiocbp equal to the address of the aiocb structure. If the aio_lio_opcode element is equal to LIO_WRITE, an I/O operation is submitted as if by a call to aio_write with the aiocbp argument equal to the address of the aiocb structure.

The aio_fildes member specifies the file descriptor on which the operation is to be performed.

The aio buf member specifies the address of the buffer to or from which the data is transferred.

The aio nbytes member specifies the number of bytes of data to be transferred.

The members of the **aiocb** structure further describe the I/O operation to be performed, in a manner identical to that of the corresponding aiocb structure when used by the aio read and aio write subroutines.

The *nent* parameter specifies how many elements are members of the list.

The behavior of the lio listio subroutine is altered according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion if synchronized I/O is enabled on the file associated with aio fildes.

For regular files, no data transfer occurs past the offset maximum established in the open file description.

Parameters

mode Determines whether the subroutine returns when the I/O operations are completed, or as soon as

the operations are queued.

An array of pointers to aio control structures defined in the aio.h file. list

Specifies the length of the array. nent

Determines when asynchronous notification occurs. sig

Execution Environment

The lio_listio and lio_listio64 subroutines can be called from the process environment only.

Return Values

EAGAIN The resources necessary to queue all the I/O requests were not available. The

application may check the error status of each aiocb to determine the individual

request(s) that failed.

The number of entries indicated by nent would cause the system-wide limit (AIO_MAX)

to be exceeded.

EINVAL The *mode* parameter is not a proper value, or the value of *nent* was greater than

AIO_LISTIO_MAX.

EINTR A signal was delivered while waiting for all I/O requests to complete during an

> LIO_WAIT operation. Since each I/O operation invoked by the lio_listio subroutine may provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited. Outstanding I/O requests are not canceled, and the application examines each list element to determine whether the

request was initiated, canceled, or completed.

EIO One or more of the individual I/O operations failed. The application may check the error

status for each aiocb structure to determine the individual request(s) that failed.

If the lio listio subroutine succeeds or fails with errors of EAGAIN, EINTR, or EIO, some of the I/O specified by the list may have been initiated. If the lio listio subroutine fails with an error code other than **EAGAIN**, **EINTR**, or **EIO**, no operations from the list were initiated. The I/O operation indicated by each list element can encounter errors specific to the individual read or write function being performed. In this event, the error status for each aiocb control block contains the associated error code. The error codes that can be set are the same as would be set by the read or write subroutines, with the following additional error codes possible:

EAGAIN The requested I/O operation was not queued due to resource limitations.

ECANCELED The requested I/O was canceled before the I/O completed due to an aio_cancel

request.

EFBIG The aio_lio_opcode argument is LIO_WRITE, the file is a regular file, aio_nbytes is

greater than 0, and aio_offset is greater than or equal to the offset maximum in the

open file description associated with aio_fildes.

EINPROGRESS The requested I/O is in progress.

The aio_lio_opcode argument is set to LIO_READ, the file is a regular file, aio_nbytes is **EOVERFLOW**

greater than 0, and the aio_offset argument is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with aio_fildes.

Related Information

"aio cancel or aio cancel64 Subroutine" on page 22, "aio error or aio error64 Subroutine" on page 25, "aio read or aio read64 Subroutine" on page 33, "aio return or aio return64 Subroutine" on page 37, "aio suspend or aio suspend64 Subroutine" on page 40, "aio write or aio write64 Subroutine" on page 43, "close Subroutine" on page 141, "exec: execl, execle, execlp, execv, execve, execvp, or exect

Subroutine" on page 196, "exit, atexit, _exit, or _Exit Subroutine" on page 203, "fork, f_fork, or vfork Subroutine" on page 248, and "Iseek, Ilseek or Iseek64 Subroutine" on page 646.

The read, ready, ready, readyx, or pread Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Legacy AIO lio_listio Subroutine

Purpose

Initiates a list of asynchronous I/O requests with a single call.

Syntax

```
#include <aio.h>
int lio listio (cmd,
list, nent, eventp)
int cmd, nent;
struct liocb * list[];
struct event * eventp;
int lio listio64
(cmd, list, nent, eventp)
int cmd, nent; struct liocb64 *list;
struct event *eventp;
```

Description

The lio_listio subroutine allows the calling process to initiate the nent parameter asynchronous I/O requests. These requests are specified in the **liocb** structures pointed to by the elements of the *list* array. The call may block or return immediately depending on the cmd parameter. If the cmd parameter requests that I/O completion be asynchronously notified, a SIGIO signal is delivered when all I/O operations are completed.

The lio_listio64 subroutine is similar to the lio_listio subroutine except that it takes an array of pointers to liocb64 structures. This allows the lio_listio64 subroutine to specify offsets in excess of OFF_MAX (2 gigbytes minus 1).

In the large file enabled programming environment, lio listio is redefined to be lio listio64.

Note: The pointer to the event structure eventp parameter is currently not in use, but is included for future compatibility.

Parameters

cmd The *cmd* parameter takes one of the following values:

LIO WAIT

Queues the requests and waits until they are complete before returning.

LIO_NOWAIT

Queues the requests and returns immediately, without waiting for them to complete. The event parameter is ignored.

LIO_NOWAIT_AIOWAIT

Queues the requests and returns immediately, without waiting for them to complete. The aio_nwait subroutine must be called for the aio control blocks to be updated.

LIO ASYNC

Queues the requests and returns immediately, without waiting for them to complete. An enhanced signal is delivered when all the operations are completed. Currently this command is not implemented.

LIO ASIG

Queues the requests and returns immediately, without waiting for them to complete. A SIGIO signal is generated when all the I/O operations are completed.

list Points to an array of pointers to **liocb** structures. The structure array contains *nent* elements:

lio aiocb

The asynchronous I/O control block associated with this I/O request. This is an actual aiocb structure, not a pointer to one.

lio fildes

Identifies the file object on which the I/O is to be performed.

lio_opcode

This field may have one of the following values defined in the /usr/include/sys/aio.h file:

LIO READ

Indicates that the read I/O operation is requested.

LIO_WRITE

Indicates that the write I/O operation is requested.

LIO_NOP

Specifies that no I/O is requested (that is, this element will be ignored).

nent

Specifies the number of entries in the array of pointers to listio structures.

eventp

Points to an **event** structure to be used when the *cmd* parameter is set to the **LIO ASYNC** value. This parameter is currently ignored.

Execution Environment

The lio listio and lio listio64 subroutines can be called from the process environment only.

Return Values

When the lio listio subroutine is successful, it returns a value of 0. Otherwise, it returns a value of -1 and sets the errno global variable to identify the error. The returned value indicates the success or failure of the lio_listio subroutine itself and not of the asynchronous I/O requests (except when the command is LIO_WAIT). The aio_error subroutine returns the status of each I/O request.

Return codes can be set to the following **errno** values:

EAGAIN

Indicates that the system resources required to queue the request are not available. Specifically, the transmit queue may be full, or the maximum number of opens may have been reached.

EFAIL Indicates that one or more I/O operations was not successful. This error can be received only if the

cmd parameter has a LIO_WAIT value.

EINTR Indicates that a signal or event interrupted the lio_listio subroutine call.

EINVAL Indicates that the aio whence field does not have a valid value or that the resulting pointer is not valid.

Related Information

The aio_cancel or aio_cancel64 ("aio_cancel or aio_cancel64 Subroutine" on page 22) subroutine, aio_error or aio_error64 ("aio_error or aio_error64 Subroutine" on page 25) subroutine, aio_read or aio_read64 ("aio_read or aio_read64 Subroutine" on page 33) subroutine, aio_return or aio_return64 ("aio return or aio return64 Subroutine" on page 37) subroutine, aio suspend or aio suspend64 ("aio_suspend or aio_suspend64 Subroutine" on page 40) subroutine, aio_write or aio_write64 ("aio_write or aio write64 Subroutine" on page 43) subroutine.

The Asynchronous I/O Overview and the Communications I/O Subsystem: Programming Introduction in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

The Input and Output Handling Programmer's Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs describes the files, commands, and subroutines used for low-level, stream, terminal, and asynchronous I/O interfaces.

Ilrint, Ilrintf, or Ilrintl Subroutine

Purpose

Rounds to the nearest integer value using current rounding direction.

Syntax

```
#include <math.h>
long long llrint (x)
double x;
long long llrintf (x)
float x:
long long llrintl (x)
long double x;
```

Description

The **Ilrint**, **Ilrintf**, and **Ilrintl** subroutines round the x parameter to the nearest integer value, rounding according to the current rounding direction.

An application wishing to check for error situations should set the **errno** global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

х Specifies the value to be rounded.

Upon successful completion, the Ilrint, Ilrintf, and Ilrintl subroutines return the rounded integer value.

If x is NaN, a domain error occurs, and an unspecified value is returned.

If x is +Inf, a domain error occurs and an unspecified value is returned.

If x is $-\ln f$, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a long long, a domain error occur and an unspecified value is returned.

If the correct value is negative and too large to represent as a long long, a domain error occurs and an unspecified value is returned.

Related Information

"feclearexcept Subroutine" on page 223 and "fetestexcept Subroutine" on page 231.

math.h in AIX 5L Version 5.2 Files Reference.

Ilround, Ilroundf, or Ilroundl Subroutine

Purpose

Rounds to the nearest integer value.

Syntax

```
#include <math.h>
long long llround (x)
double x;
long long llroundf (x)
float x;
long long llroundl (x)
long double x;
```

Description

The **Ilround**, **Ilroundf**, or **Ilroundl** subroutines round the x parameter to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

An application wishing to check for error situations should set the **errno** global variable to zero and call feclearexcept(FE ALL EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be rounded.

Return Values

Upon successful completion, the Ilround, Ilroundf, or IlroundI subroutines return the rounded integer value.

If x is NaN, a domain error occurs, and an unspecified value is returned.

If x is +Inf, a domain error occurs and an unspecified value is returned.

If x is -Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a long long, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a long long, a domain error occurs and an unspecified value is returned.

Related Information

"feclearexcept Subroutine" on page 223 and "fetestexcept Subroutine" on page 231.

math.h in AIX 5L Version 5.2 Files Reference.

load Subroutine

Purpose

Loads a module into the current process.

Syntax

```
int *load ( ModuleName, Flags, LibraryPath)
char *ModuleName;
uint Flags;
char *LibraryPath;
```

Description

The **load** subroutine loads the specified module into the calling process's address space. A module can be a regular file or a member of an archive. When adding a new module to the address space of a 32-bit process, the load operation may cause the break value to change.

The **exec** subroutine is similar to the **load** subroutine, except that:

- The **load** subroutine does not replace the current program with a new one.
- The exec subroutine does not have an explicit library path parameter; it has only the LIBPATH environment variable. Also, the LIBPATH variable is ignored when the program using the exec subroutine has more privilege than the caller, for example, in the case of a set-UID program.

A large application can be split up into one or more modules in one of two ways that allow execution within the same process. The first way is to create each of the application's modules separately and use **load** to explicitly load a module when it is needed. The other way is to specify the relationship between the modules when they are created by defining imported and exported symbols.

Modules can import symbols from other modules. Whenever symbols are imported from one or more other modules, these modules are automatically loaded to resolve the symbol references if the required modules are not already loaded, and if the imported symbols are not specified as deferred imports. These modules can be archive members in libraries or individual files and can have either shared or private file characteristics that control how and where they are loaded.

Shared modules (typically members of a shared library archive) are loaded into the shared library region, when their access permissions allow sharing, that is, when they have read-other permission. Private modules, and shared modules without the required permissions for sharing, are loaded into the process private region.

When the loader resolves a symbol, it uses the file name recorded with that symbol to find the module that exports the symbol. If the file name contains any / (slash) characters, it is used directly and must name an appropriate file or archive member. However, if the file name is a base name (contains no / characters), the loader searches the directories specified in the default library path for a file (i.e. a module or an archive) with that base name.

The LibraryPath is a string containing one or more directory path names separated by colons. See the section "Searching for Dependent Modules" for information on library path searching.

(This paragraph only applies to AIX 4.3.1 and previous releases.) When a process is executing under ptrace control, portions of the process's address space are recopied after the load processing completes. For a 32-bit process, the main program text (loaded in segment 1) and shared library modules (loaded in segment 13) are recopied. Any breakpoints or other modifications to these segments must be reinserted after the load call. For a 64-bit process, shared library modules are recopied after a load call. The debugger will be notified by setting the W SLWTED flag in the status returned by wait, so that it can reinsert breakpoints.

(This paragraph only applies to AIX 4.3.2 and later releases.) When a process executing under **ptrace** control calls load, the debugger is notified by setting the W SLWTED flag in the status returned by wait. Any modules newly loaded into the shared library segments will be copied to the process's private copy of these segments, so that they can be examined or modified by the debugger.

If the program calling the load subroutine was linked on AIX 4.2 or a later release, the load subroutine will call initialization routines (init routines) for the new module and any of its dependents if they were not already loaded.

Modules loaded by this subroutine are automatically unloaded when the process terminates or when the exec subroutine is executed. They are explicitly unloaded by calling the unload subroutine.

Searching for Dependent Modules

The load operation and the exec operation differ slightly in their dependent module search mechanism. When a module is added to the address space of a running process (the load operation), the rules outlined in the next section are used to find the named module. Note that dependency relationships may be loosely defined as a tree but recursive relationships between modules may also exist. The following components may used to create a complete library search path:

- 1. If the L LIBPATH EXEC flag is set, the library search path used at exec-time.
- 2. The value of the LibraryPath parameter if it is non-null. Note that a null string is a valid search path which refers to the current working directory. If the LibraryPath parameter is NULL, the value of the LIBPATH environment variable is used instead.
- 3. The library search path contained in the loader section of the module being loaded (the ModuleName parameter).
- 4. The library search path contained in the loader section of the module whose immediate dependents are being loaded. Note that this per-module information changes when searching for each module's immediate dependents.

To find the ModuleName module, components 1 and 2 are used. To find dependents, components 1, 2, 3 and 4 are used in order. Note that if any modules that are already part of the running process satisfy the dependency requirements of the newly loaded module(s), pre-existing modules are not loaded again.

For each colon-separated portion of the aggregate search specification, if the base name is not found the search continues. The first instance of the base name found is used; if the file is not of the proper form, or in the case of an archive does not contain the required archive member, or does not export a definition of a required symbol, an error occurs. The library path search is not performed when either a relative or an absolute path name is specified for a dependent module.

The library search path stored within the module is specified at link-edit time.

The load subroutine may cause the calling process to fail if the module specified has a very long chain of dependencies, (for example, lib1.a, which depends on lib2.a, which depends on lib3.a, etc). This is because the loader processes such relationships recursively on a fixed-size stack. This limitation is exposed only when processing a dependency chain that has over one thousand elements.

Parameters

ModuleName

Points to the name of the module to be loaded. The module name consists of a path name, and, an optional member name. If the path name contains at least on / character, the name is used directly, and no directory searches are performed to locate the file. If the path name contains no / characters, it is treated as a base name, and should be in one of the directories listed in the library path.

The library path is either the value of the LibraryPath parameter if not a null value, or the value of the LIBPATH environment variable (if set) or the library path used at process exec time (if the L LIBPATH EXEC is set). If no library path is provided, the module should be in the current directory.

The ModuleName parameter may explicitly name an archive member. The syntax is pathname(member) where pathname follows the rules specified in the previous paragraph, and member is the name of a specific archive member. The parentheses are a required portion of the specification and no intervening spaces are allowed. If an archive member is named, the L_LOADMEMBER flag must be added to the Flags parameter. Otherwise, the entire ModuleName parameter is treated as an explicit filename.

Modifies the behavior of the load service as follows (see the ldr.h file). If no special behavior is required, set the value of the flags parameter to 0 (zero). For compatibility, a value of 1 (one) may also be specified.

L_LIBPATH_EXEC

Specifies that the library path used at process exec time should be prepended to any library path specified in the load call (either as an argument or environment variable). It is recommended that this flag be specified in all calls to the **load** subroutine.

L LOADMEMBER

Indicates that the ModuleName parameter may specify an archive member. The ModuleName argument is searched for parentheses, and if found the parameter is treated as a filename/member name pair. If this flag is present and the ModuleName parameter does not contain parenthesis the entire ModuleName parameter is treated as a filename specification. Under either condition the filename is expected to be found within the library path or the current directory.

L NOAUTODEFER

Specifies that any deferred imports in the module being loaded must be explicitly resolved by use of the loadbind subroutine. This allows unresolved imports to be explicitly resolved at a later time with a specified module. If this flag is not specified, deferred imports (marked for deferred resolution) are resolved at the earliest opportunity when any subsequently loaded module exports symbols matching unresolved imports.

Flags

LibraryPath Points to a character string that specifies the default library search path.

> If the LibraryPath parameter is NULL the LIBPATH environment variable is used. See the section "Searching for Dependent Modules" on page 619 for more information.

The library path is used to locate dependent modules that are specified as basenames (that is, their pathname components do not contain a / (slash) character.

Note the difference between setting the LibraryPath parameter to null, and having the LibraryPath parameter point to a null string (" "). A null string is a valid library path which consists of a single directory: the current directory.

Return Values

Upon successful completion, the load subroutine returns the pointer to function for the entry point of the module. If the module has no entry point, the address of the data section of the module is returned.

Error Codes

If the **load** subroutine fails, a null pointer is returned, the module is not loaded, and **errno** global variable is set to indicate the error. The load subroutine fails if one or more of the following are true of a module to be explicitly or automatically loaded:

EACCES Indicates the file is not an ordinary file, or the mode of the program file denies execution

permission, or search permission is denied on a component of the path prefix.

EINVAL Indicates the file or archive member has a valid magic number in its header, but the header is

damaged or is incorrect for the machine on which the file is to be run.

ELOOP Indicates too many symbolic links were encountered in translating the path name.

ENOEXEC Indicates an error occurred when loading or resolving symbols for the specified module. This

can be due to an attempt to load a module with an invalid XCOFF header, a failure to resolve

symbols that were not defined as deferred imports or several other load time related problems. The loadquery subroutine can be used to return more information about the load failure. If the main program was linked on a AIX 4.2 or later system, and if runtime linking is used, the load subroutine will fail if the runtime linker could not resolve some symbols. In this

case, errno will be set to ENOEXEC, but the loadquery subroutine will not return any

additional information.

ENOMEM Indicates the program requires more memory than is allowed by the system-imposed

ETXTBSY Indicates the file is currently open for writing by some process.

ENAMETOOLONG Indicates a component of a path name exceeded 255 characters, or an entire path name

exceeded 1023 characters.

ENOENT Indicates a component of the path prefix does not exist, or the path name is a null value. For

the **dlopen** subroutine, RTLD MEMBER is not used when trying to open a member within

the archive file.

ENOTDIR Indicates a component of the path prefix is not a directory.

ESTALE Indicates the process root or current directory is located in a virtual file system that has been

unmounted.

Related Information

The **dlopen** ("dlopen Subroutine" on page 178) subroutine, **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutine, **loadbind** ("loadbind Subroutine" on page 622) subroutine, loadquery ("loadquery Subroutine" on page 623) subroutine, ptrace ("ptrace, ptracex, ptrace64 Subroutine" on page 978) subroutine, unload subroutine.

The Id command.

The Shared Library Overview and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

loadbind Subroutine

Purpose

Provides specific run-time resolution of a module's deferred symbols.

Syntax

int loadbind(Flag, ExportPointer, ImportPointer) int Flaa: void *ExportPointer, *ImportPointer;

Description

The loadbind subroutine controls the run-time resolution of a previously loaded object module's unresolved imported symbols.

The loadbind subroutine is used when two modules are loaded. Module A, an object module loaded at run time with the load subroutine, has designated that some of its imported symbols be resolved at a later time. Module B contains exported symbols to resolve module A's unresolved imports.

To keep module A's imported symbols from being resolved until the loadbind service is called, you can specify the load subroutine flag, L_NOAUTODEFER, when loading module A.

(This paragraph only applies to AIX 4.3.1 and previous releases.) When a 32-bit process is executing under ptrace control, portions of the process's address space are recopied after the loadbind processing completes. The main program text (loaded in segment 1) and shared library modules (loaded in segment 13) are recopied. Any breakpoints or other modifications to these segments must be reinserted after the loadbind call.

(This paragraph only applies to AIX 4.3.2 and later releases.) When a 32-bit process executing under ptrace control calls loadbind, the debugger is notified by setting the W_SLWTED flag in the status returned by wait.

When a 64-bit process under ptrace control calls loadbind, the debugger is not notified and execution of the process being debugged continues normally.

Parameters

Flag Currently not used.

ExportPointer Specifies the function pointer returned by the load subroutine when module B was loaded. ImportPointer Specifies the function pointer returned by the load subroutine when module A was loaded.

Note: The ImportPointer or ExportPointer parameter may also be set to any exported static data area symbol or function pointer contained in the associated module. This would typically be the function pointer returned from the load of the specified module.

Return Values

A 0 is returned if the **loadbind** subroutine is successful.

Error Codes

A -1 is returned if an error is detected, with the errno global variable set to an associated error code:

EINVAL Indicates that either the ImportPointer or ExportPointer parameter is not valid (the pointer to the

ExportPointer or ImportPointer parameter does not correspond to a loaded program module or library).

ENOMEM Indicates that the program requires more memory than allowed by the system-imposed maximum.

After an error is returned by the loadbind subroutine, you may also use the loadquery subroutine to obtain additional information about the loadbind error.

Related Information

The load ("load Subroutine" on page 618) subroutine, loadquery ("loadquery Subroutine") subroutine, unload subroutine.

The Id command.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

loadquery Subroutine

Purpose

Returns error information from the **load** or **exec** subroutine; also provides a list of object files loaded for the current process.

Syntax

int loadquery(Flags, Buffer, BufferLength) int Flags; void *Buffer; unsigned int BufferLength;

Description

The loadquery subroutine obtains detailed information about an error reported on the last load or exec subroutine executed by a calling process. The loadquery subroutine may also be used to obtain a list of object file names for all object files that have been loaded for the current process, or the library path that was used at process exec time.

Parameters

Buffer Points to a Buffer in which to store the information.

BufferLength Specifies the number of bytes available in the Buffer parameter. Flags

Specifies the action of the loadquery subroutine as follows:

L_GETINFO

Returns a list of all object files loaded for the current process, and stores the list in the Buffer parameter. The object file information is contained in a sequence of LD_INFO structures as defined in the sys/ldr.h file. Each structure contains the module location in virtual memory and the path name that was used to load it into memory. The file descriptor field in the **LD_INFO** structure is not filled in by this function.

L_GETMESSAGE

Returns detailed error information describing the failure of a previously invoked load or exec function, and stores the error message information in Buffer. Upon successful return from this function the beginning of the Buffer contains an array of character pointers. Each character pointer points to a string in the buffer containing a loader error message. The character array ends with a null character pointer. Each error message string consists of an ASCII message number followed by zero or more characters of error-specific message data. Valid message numbers are listed in the sys/ldr.h file.

You can format the error messages returned by the L_GETMESSAGE function and write them to standard error using the standard system command /usr/sbin/execerror as follows:

```
char *buffer[1024];
buffer[0] = "execerror";
buffer[1] = "name of program that failed to load";
loadquery(L GETMESSAGES, &buffer[2],\
  sizeof buffer-2*sizeof(char*));
execvp("/usr/sbin/execerror",buffer);
```

This sample code causes the application to terminate after the messages are written to standard error.

L GETLIBPATH

Returns the library path that was used at process exec time. The library path is a null terminated character string.

Return Values

Upon successful completion, **loadquery** returns the requested information in the caller's buffer specified by the Buffer and BufferLength parameters.

Error Codes

The **loadquery** subroutine returns with a return code of -1 and the **errno** global variable is set to one of the following when an error condition is detected:

ENOMEM Indicates that the caller's buffer specified by the Buffer and BufferLength parameters is too small to return the information requested. When this occurs, the information in the buffer is undefined.

EINVAL Indicates the function specified in the Flags parameter is not valid.

EFAULT Indicates the address specified in the Buffer parameter is not valid.

Related Information

The exec ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutine, load ("load Subroutine" on page 618) subroutine, loadbind ("loadbind Subroutine" on page 622) subroutine, unload subroutine.

The Id command.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

localecony Subroutine

Purpose

Sets the locale-dependent conventions of an object.

Library

Standard C Library (libc.a)

Syntax

```
#include <locale.h>
struct lconv *localeconv ( )
```

Description

The **localeconv** subroutine sets the components of an object using the **lconv** structure. The **lconv** structure contains values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The fields of the structure with the type **char** * are strings, any of which (except decimal_point) can point to a null string, which indicates that the value is not available in the current locale or is of zero length. The fields with type **char** are nonnegative numbers, any of which can be the **CHAR_MAX** value which indicates that the value is not available in the current locale. The fields of the **lconv** structure include the following:

```
char *decimal_point
char *thousands_sep
char *grouping
```

The decimal-point character used to format non-monetary quantities. The character used to separate groups of digits to the left of the decimal point in formatted non-monetary quantities.

A string whose elements indicate the size of each group of digits in formatted non-monetary quantities.

The value of the grouping field is interpreted according to the following:

CHAR_MAX

No further grouping is to be performed.

The previous element is to be repeatedly used for the remainder of the digits.

other

The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

The international currency symbol applicable to the current locale, left-justified within a four-character space-padded field. The character sequences are in accordance with those specified in ISO 4217, "Codes for the Representation of Currency and Funds."

The local currency symbol applicable to the current locale. The decimal point used to format monetary quantities.

The separator for groups of digits to the left of the decimal point in formatted monetary quantities.

char *int_curr_symbol

char *currency_symbol
char *mon_decimal_point
char *mon thousands sep

char *mon grouping

char *positive_sign
char *negative_sign
char int_frac_digits

char p_cs_precedes

char p_sep_by_space

char n cs precedes

char n_sep_by_space

char p sign posn

char n sign posn

A string whose elements indicate the size of each group of digits in formatted monetary quantities.

The value of the mon_grouping field is interpreted according to the following:

CHAR MAX

No further grouping is to be performed.

The previous element is to be repeatedly used for the remainder of the digits.

other The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

The string used to indicate a nonnegative formatted monetary quantity. The string used to indicate a negative formatted monetary quantity. The number of fractional digits (those to the right of the decimal point) to be displayed in a formatted monetary quantity.

Set to 1 if the specified currency symbol (the currency_symbol or int_curr_symbol field) precedes the value for a nonnegative formatted monetary quantity and set to 0 if the specified currency symbol follows the value for a nonnegative formatted monetary quantity.

Set to 1 if the currency_symbol or int_curr_symbol field is separated by a space from the value for a nonnegative formatted monetary quantity and set to 0 if the currency_symbol or int_curr_symbol field is not separated by a space from the value for a nonnegative formatted monetary quantity.

Set to 1 if the currency_symbol or int_curr_symbol field precedes the value for a negative formatted monetary quantity and set to 0 if the currency_symbol or int_curr_symbol field follows the value for a negative formatted monetary quantity.

Set to 1 if the currency_symbol or int_curr_symbol field is separated by a space from the value for a negative formatted monetary quantity and set to 0 if the currency_symbol or int_curr_symbol field is not separated by a space from the value for a negative formatted monetary quantity. Set to 2 if the symbol and the sign string are adjacent and separated by a blank character.

Set to a value indicating the positioning of the positive sign (the positive_sign fields) for nonnegative formatted monetary quantity. Set to a value indicating the positioning of the negative sign (the negative_sign fields) for a negative formatted monetary quantity.

The values of the p_sign_posn and n_sign_posn fields are interpreted according to the following definitions:

- Parentheses surround the quantity and the specified currency symbol or international currency symbol.
- 1 The sign string precedes the quantity and the currency symbol or international currency symbol.
- 2 The sign string follows the quantity and currency symbol or international currency symbol.
- 3 The sign string immediately precedes the currency symbol or international currency symbol.
- The sign string immediately follows the currency symbol or international currency symbol.

The following table illustrates the rules that can be used by three countries to format monetary quantities:

Country	Positive Format	Negative Format	International Format
Italy	L.1234	-L.1234	ITL.1234
Norway	krl.234.56	krl.234.56-	NOK 1.234.56
Switzerland	SFrs.1.234.56	SFrs.1.234.56C	CHF 1.234.56

The following table shows the values of the monetary members of the structure returned by the localecony subroutine for these countries:

struct localeconv	Italy	Norway	Switzerland
char *in_curr_symbol	"ITL."	"NOK"	"CHF"
char *currency_symbol	"L."	"kr"	"SFrs."
char *mon_decimal_point	" "	"."	"."
char *mon_thousands_sep	"."	"."	"."
char *mon_grouping	"\3"	"\3"	"\3"
char *positive_sign	" "	" "	" "
char *negative_sign	"_"	"_"	"C"
char int_frac_digits	0	2	2
char frac_digits	0	2	2
char p_cs_precedes	1	1	1
char p_sep_by_space	0	0	0
char n_cs_precedes	1	1	1
char n_sep_by_space	0	0	0
char p_sign_posn	1	1	1
char n_sign_posn	1	2	2
			<u> </u>

Return Values

A pointer to the filled-in object is returned. In addition, calls to the setlocale subroutine with the LC ALL, LC_MONETARY or LC_NUMERIC categories may cause subsequent calls to the localeconv subroutine to return different values based on the selection of the locale.

Note: The structure pointed to by the return value is not modified by the program but may be overwritten by a subsequent call to the **localeconv** subroutine.

Related Information

The "nl_langinfo Subroutine" on page 736, rpmatch subroutine, setlocale subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview and Setting the Locale in AIX 5L Version 5.2 National Language Support Guide and Reference.

lockfx, lockf, flock, or lockf64 Subroutine

Purpose

Locks and unlocks sections of open files.

Libraries

lockfx, lockf: Standard C Library (libc.a)

flock: Berkeley Compatibility Library (libbsd.a)

Syntax

#include <fcntl.h>

```
int lockfx (FileDescriptor,
Command, Argument)
int FileDescriptor;
int Command;
struct flock * Argument;
#include <sys/lockf.h>
#include <unistd.h>
int lockf
(FileDescriptor, Request, Size)
int FileDescriptor;
int Request;
off_t Size;
int lockf64 (FileDescriptor,
Request, Size)
int FileDescriptor;
int Request;
off64_t Size;
#include <sys/file.h>
int flock (FileDescriptor, Operation)
int FileDescriptor;
int Operation;
```

Description

Attention: Buffered I/O does not work properly when used with file locking. Do not use the standard I/O package routines on files that are going to be locked.

The **lockfx** subroutine locks and unlocks sections of an open file. The **lockfx** subroutine provides a subset of the locking function provided by the **fcntl** ("fcntl, dup, or dup2 Subroutine" on page 214) subroutine.

The **lockf** subroutine also locks and unlocks sections of an open file. However, its interface is limited to setting only write (exclusive) locks.

Although the **lockfx**, **lockf**, **flock**, and **fcntl** interfaces are all different, their implementations are fully integrated. Therefore, locks obtained from one subroutine are honored and enforced by any of the lock subroutines.

The Operation parameter to the **lockfx** subroutine, which creates the lock, determines whether it is a read lock or a write lock.

The file descriptor on which a write lock is being placed must have been opened with write access.

lockf64 is equivalent to **lockf** except that a 64-bit lock request size can be given. For **lockf**, the largest value which can be used is OFF_MAX, for lockf64, the largest value is LONGLONG_MAX.

In the large file enabled programming environment, lockf is redefined to be lock64.

The flock subroutine locks and unlocks entire files. This is a limited interface maintained for BSD compatibility, although its behavior differs from BSD in a few subtle ways. To apply a shared lock, the file must be opened for reading. To apply an exclusive lock, the file must be opened for writing.

Locks are not inherited. Therefore, a child process cannot unlock a file locked by the parent process.

Parameters

Argument Command A pointer to a structure of type flock, defined in the flock.h file. Specifies one of the following constants for the **lockfx** subroutine:

F_SETLK

Sets or clears a file lock. The 1_type field of the **flock** structure indicates whether to establish or remove a read or write lock. If a read or write lock cannot be set, the lockfx subroutine returns immediately with an error value of -1.

F_SETLKW

Performs the same function as F_SETLK unless a read or write lock is blocked by existing locks. In that case, the process sleeps until the section of the file is free to be locked.

F GETLK

Gets the first lock that blocks the lock described in the flock structure. If a lock is found, the retrieved information overwrites the information in the flock structure. If no lock is found that would prevent this lock from being created, the structure is passed back unchanged except that the 1_type field is set to F_UNLCK.

FileDescriptor

A file descriptor returned by a successful open or fcntl subroutine, identifying the file to which the lock is to be applied or removed.

Operation

Specifies one of the following constants for the **flock** subroutine:

LOCK_SH

Apply a shared (read) lock.

LOCK EX

Apply an exclusive (write) lock.

LOCK_NB

Do not block when locking. This value can be logically ORed with either LOCK SH or LOCK EX.

LOCK UN

Remove a lock.

Request

Specifies one of the following constants for the lockf subroutine:

F_ULOCK

Unlocks a previously locked region in the file.

F LOCK

Locks the region for exclusive (write) use. This request causes the calling process to sleep if the requested region overlaps a locked region, and to resume when granted the lock.

F_TEST

Tests to see if another process has already locked a region. The lockf subroutine returns 0 if the region is unlocked. If the region is locked, then -1 is returned and the errno global variable is set to EACCES.

F TLOCK

Locks the region for exclusive use if another process has not already locked the region. If the region has already been locked by another process, the lockf subroutine returns a -1 and the errno global variable is set to EACCES.

Size

The number of bytes to be locked or unlocked for the **lockf** subroutine. The region starts at the current location in the open file, and extends forward if the Size value is positive and backward if the Size value is negative. If the Size value is 0, the region starts at the current location and extends forward to the maximum possible file size, including the unallocated space after the end of the file.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The lockfx, lockf, and flock subroutines fail if one of the following is true:

EBADF The *FileDescriptor* parameter is not a valid open file descriptor.

EINVAL The function argument is not one of F_LOCK, F_TLOCK, F_TEST or F_ULOCK; or size plus the

current file offset is less than 0.

EINVAL An attempt was made to lock a fifo or pipe.

The lock is blocked by a lock from another process. Putting the calling process to sleep while **EDEADLK**

waiting for the other lock to become free would cause a deadlock.

ENOLCK The lock table is full. Too many regions are already locked.

EINTR The command parameter was **F_SETLKW** and the process received a signal while waiting to

acquire the lock.

EOVERFLOW The offset of the first, or if size is not 0 then the last, byte in the requested section cannot be

represented correctly in an object of type off_t.

The **lockfx** and **lockf** subroutines fail if one of the following is true:

EACCES The Command parameter is F_SETLK, the 1 type field is F_RDLCK, and the segment of the file to be

locked is already write-locked by another process.

EACCES The Command parameter is F_SETLK, the 1 type field is F_WRLCK, and the segment of a file to be

locked is already read-locked or write-locked by another process.

The **flock** subroutine fails if the following is true:

EWOULDBLOCK The file is locked and the LOCK_NB option was specified.

Related Information

The close ("close Subroutine" on page 141) subroutine, exec: execl, execv, execle, execlp, execvp, or exect ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutine, fcntl ("fcntl, dup, or dup2 Subroutine" on page 214) subroutine, fork ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, open, openx, or creat ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

log10, log10f, or log10l Subroutine

Purpose

Computes the Base 10 logarithm.

Syntax

```
#include <math.h>
float log10f (x)
float x;
long double log101 (x)
long double x;
double log10 (x)
double x;
```

Description

The **log10f**, **log10f**, and **log10** subroutines compute the base 10 logarithm of the x parameter, $\log_{10}(x)$.

An application wishing to check for error situations should set errno to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the log10, log10f, and log10l, subroutines return the base 10 logarithm of x.

If x is ± 0 , a pole error occurs and log10, log10f, and log10l subroutines return -HUGE_VAL, -HUGE_VALF and -HUGE_VALL, respectively.

For finite values of x that are less than 0, or if x is -Inf, a domain error occurs, and a NaN is returned.

If x is NaN, a NaN is returned.

If x is 1, +0 is returned.

If x is +Inf, +Inf is returned.

Error Codes

When using the libm.a library:

log10 If the x parameter is less than 0, the log10 subroutine returns a NaNQ value and sets errno to EDOM.

If x= 0, the log10 subroutine returns a -HUGE_VAL value but does not modify errno.

When using libmsaa.a(-lmsaa):

log10 If the x parameter is not positive, the log10 subroutine returns a -HUGE_VAL value and sets errno

to **EDOM**. A message indicating DOMAIN error (or SING error when x = 0) is output to standard

error.

log10 If x < 0, log10I returns the value NaNQ and sets errno to EDOM. If x = 0, log10I returns the

value -HUGE_VAL but does not modify errno.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, "class, _class, finite, isnan, or unordered Subroutines" on page 138, and "madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout, omout, fmout, m_out, sdiv, or itom Subroutine" on page 661.

math.h in AIX 5L Version 5.2 Files Reference.

log1p, log1pf, or log1pl Subroutine

Purpose

Computes a natural logarithm.

Syntax

```
#include <math.h>
float log1pf (x)
float x;
long double log1pl (x)
long double x;
double log1p (x)
double x;
```

Description

The **log1pf**, **log1pl**, and **log1p** subroutines compute $log_e (1.0 + x)$.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

x Specifies the value to be computed.

Return Values

Upon successful completion, the log1pf, log1pl, and log1p subroutines return the natural logarithm of 1.0 + X.

If x is -1, a pole error occurs and the log1pf, log1pl, and log1p subroutines return -HUGE_VALF, -HUGE_VALL, and -HUGE_VAL, respectively.

For finite values of x that are less than -1, or if x is -Inf, a domain error occurs, and a NaN is returned.

If x is NaN, a NaN is returned.

If x is ± 0 , or $+ \ln f$, x is returned.

If x is subnormal, a range error may occur and x should be returned.

Related Information

"feclearexcept Subroutine" on page 223 and "fetestexcept Subroutine" on page 231.

math.h in AIX 5L Version 5.2 Files Reference.

log2, log2f, or log2l Subroutine

Purpose

Computes base 2 logarithm.

Syntax

```
#include <math.h>
double log2(x)
double x;
float log2f (x)
float x;
long double log21 (x)
long double x;
```

Description

The log2, log2f, and log2l subroutines compute the base 2 logarithm of the x parameter, $log_2(x)$.

An application wishing to check for error situations should set errno to zero and call feclearexcept(FE ALL EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the log2, log2f, and log2l subroutines return the base 2 logarithm of x.

If x is ± 0 , a pole error occurs and the $\log 2$, $\log 2f$, and $\log 2I$ subroutines return -HUGE_VAL, -HUGE_VALF, and -HUGE_VALL, respectively.

For finite values of x that are less than 0, or if x is -Inf, a domain error occurs, and a NaN is returned.

If x is NaN, a NaN is returned.

If x is 1, +0 is returned.

If x is +Inf, x is returned.

Related Information

"feclearexcept Subroutine" on page 223 and "fetestexcept Subroutine" on page 231.

math.h in AIX 5L Version 5.2 Files Reference.

logbf, logbl, or logb Subroutine

Purpose

Computes the radix-independent exponent.

Syntax

```
#include <math.h>
float logbf (x)
float x;
long double logbl (x)
long double x;
double logb(x)
double x;
```

Description

The **logbf** and **logbl** subroutines compute the exponent of x, which is the integral part of $\log_r |x|$, as a signed floating-point value, for nonzero x, where r is the radix of the machine's floating-point arithmetic. For AIX, FLT RADIX r=2.

```
If x is subnormal, it is treated as though it were normalized; thus for finite positive x:

1 <= x * FLT RADIX^{-logb(x)} < FLT RADIX
```

An application wishing to check for error situations should set **errno** to zero and call **feclearexcept(FE_ALL_EXCEPT)** before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Note: When the x parameter is finite and not zero, the **logb** (x) subroutine satisfies the following equation: $1 < = \text{scalb}(|x|, -(\text{int}) \log b(x)) < 2$

Parameters

x Specifies the value to be computed.

Return Values

Upon successful completion, the **logbf** and **logbl** subroutines return the exponent of x.

If $x ext{ is } \pm 0$, a pole error occurs and the **logbf** and **logbl** subroutines return -**HUGE_VALF** and -HUGE_VALL, respectively.

If x is NaN, a NaN is returned.

If x is $\pm \ln f$, $+\ln f$ is returned.

Error Codes

The logb function returns -HUGE_VAL when the x parameter is set to a value of 0 and sets errno to EDOM.

Related Information

"feclearexcept Subroutine" on page 223 and "fetestexcept Subroutine" on page 231.

math.h in AIX 5L Version 5.2 Files Reference.

log, logf, or logI Subroutine

Purpose

Computes the natural logarithm.

Syntax

```
#include <math.h>
float logf (x)
float x;
long double log1 (x)
long double x;
double log (x)
double x;
```

Description

The **logf**, **logf**, and **log** subroutines compute the natural logarithm of the x parameter, $log_e(x)$.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE ALL EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed.

Return Values

Upon successful completion, the **logf**, **logl**, and **log** subroutines return the natural logarithm of x.

If x is ± 0 , a pole error occurs and the **logf**, **logI**, and **log** subroutines return **-HUGE_VALF** and **-HUGE VAL**, respectively.

For finite values of x that are less than 0, or if x is -Inf, a domain error occurs, and a NaN is returned.

If x is NaN, a NaN is returned.

If x is 1, +0 is returned.

If x is +Inf, x is returned.

Error Codes

When using the libm.a library:

log If the x parameter is less than 0, the log subroutine returns a NaNQ value and sets errno to EDOM. If

x= 0, the log subroutine returns a -HUGE_VAL value but does not modify errno.

When using libmsaa.a(-lmsaa):

log If the x parameter is not positive, the log subroutine returns a -HUGE_VAL value, and sets errno to

a **EDOM** value. A message indicating DOMAIN error (or SING error when x = 0) is output to

standard error.

log If x<0, the logI subroutine returns a NaNQ value

Related Information

"exp, expf, or expl Subroutine" on page 205, "feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, "class, _class, finite, isnan, or unordered Subroutines" on page 138, and "log10, log10f, or log10l Subroutine" on page 631.

math.h in AIX 5L Version 5.2 Files Reference.

loginfailed Subroutine

Purpose

Records an unsuccessful login attempt.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
int loginfailed ( User, Host, Tty, Reason)
char *User;
char *Host;
char *Tty;
int Reason;
```

Note: This subroutine is not thread-safe.

unsuccessful_login_count

The **loginfailed** subroutine performs the processing necessary when an unsuccessful login attempt occurs. If the specified user name is not valid, the UNKNOWN_USER value is substituted for the user name. This substitution prevents passwords entered as the user name from appearing on screen.

The following attributes in /etc/security/lastlog file are updated for the specified user, if the user name is valid:

time_last_unsuccessful_login Contains the current time. tty last unsuccessful login Contains the value specified by the *Tty* parameter. host_last_unsuccessful_login Contains the value specified by the Host parameter, or the

local hostname if the Host parameter is a null value. Indicates the number of unsuccessful login attempts. The loginfailed subroutine increments this attribute by one for

each failed attempt.

A login failure audit record is cut to indicate that an unsuccessful login attempt occurred. A utmp entry is appended to /etc/security/failedlogin file, which tracks all failed login attempts.

If the current unsuccessful login and the previously recorded unsuccessful logins constitute too many unsuccessful login attempts within too short of a time period (as specified by the logindisable and logininterval port attributes), the port is locked. When a port is locked, a PORT_Locked audit record is written to inform the system administrator that the port has been locked.

If the login retry delay is enabled (as specified by the **logindelay** port attribute), a sleep occurs before this subroutine returns. The length of the sleep (in seconds) is determined by the logindelay value multiplied by the number of unsuccessful login attempts that occurred in this process.

Parameters

User Specifies the user's login name who has unsuccessfully attempted to login.

Specifies the name of the host from which the user attempted to login. If the Host parameter is Null, the Host

name of the local host is used.

Tty Specifies the name of the terminal on which the user attempted to login.

Reason Specifies a reason code for the login failure. Valid values are AUDIT FAIL and AUDIT FAIL AUTH

defined in the sys/audit.h file.

Security

Access Control: The calling process must have access to the account information in the user database and the port information in the port database.

File Accessed:

Mode File /etc/security/user /etc/security/lastlog rw /etc/security/login.cfg /etc/security/portlog rw /etc/security/failedlogin w

Auditing Events:

Event Information
USER_Login username
PORT_Locked portname

Return Values

Upon successful completion, the **loginfailed** subroutine returns a value of 0. If an error occurs, a value of -1 is returned and error is set to indicate the error.

Error Codes

The loginfailed subroutine fails if one or more of the following values is true:

EACCES The current process does not have access to the user or port database. **EPERM** The current process does not have permission to write an audit record.

Related Information

The **authenticate** ("authenticate Subroutine" on page 94) subroutine, **getpcred** ("getpcred Subroutine" on page 341) subroutine, **getpenv** ("getpenv Subroutine" on page 343) subroutine, **loginrestrictions** ("loginrestrictions Subroutine") subroutine, **loginsuccess** ("loginsuccess Subroutine" on page 641) subroutine, **setpcred** subroutine, **setpenv** subroutine.

List of Security and Auditing Services in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

loginrestrictions Subroutine

Purpose

Determines if a user is allowed to access the system.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
#include <login.h>

int loginrestrictions (Name, Mode, Tty, Msg)
char * Name;
int Mode;
char * Tty;
char ** Msg;
```

Note: This subroutine is not thread-safe.

The **loginrestrictions** subroutine determines if the user specified by the *Name* parameter is allowed to access the system. The Mode parameter gives the mode of account usage and the Tty parameter defines the terminal used for access. The Msg parameter returns an informational message explaining why the **loginrestrictions** subroutine failed.

This subroutine is unsuccessful if any of the following conditions exists:

- The user's account has expired as defined by the expires user attribute.
- The user's account has been locked as defined by the account_locked user attribute.
- The user attempted too many unsuccessful logins as defined by the loginretries user attribute.
- The user is not allowed to access the given terminal as defined by the ttys user attribute.
- The user is not allowed to access the system at the present time as defined by the logintimes user attribute.
- The Mode parameter is set to the S LOGIN value or the S RLOGIN value, and too many users are logged in as defined by the maxlogins system attribute.
- The Mode parameter is set to the S_LOGIN value and the user is not allowed to log in as defined by the **login** user attribute.
- The Mode parameter is set to the S_RLOGIN value and the user is not allowed to log in from the network as defined by the rlogin user attribute.
- The Mode parameter is set to the S SU value and other users are not allowed to use the su command as defined by the su user attribute, or the group ID of the current process cannot use the su command to switch to this user as defined by the **sugroups** user attribute.
- The Mode parameter is set to the S DAEMON value and the user is not allowed to run processes from the **cron** or **src** subsystem as defined by the **daemon** user attribute.
- The terminal is locked as defined by the **locktime** port attribute.
- The user cannot use the terminal to access the system at the present time as defined by the logintimes port attribute.
- The user is not the root user and the /etc/nologin file exists.

Note: The loginrestrictions subroutine is not safe in a multi-threaded environment. To use loginrestrictions in a threaded application, the application must keep the integrity of each thread.

Parameters

Name Specifies the user's login name whose account is to be validated.

Specifies the mode of usage. Valid values as defined in the login.h file are listed below. The Mode Mode parameter has a value of 0 or one of the following values:

S_LOGIN

Verifies that local logins are permitted for this account.

Verifies that the su command is permitted and the current process has a group ID that can invoke the **su** command to switch to the account.

S DAEMON

Verifies the account can invoke daemon or batch programs through the **src** or **cron** subsystems.

S RLOGIN

Verifies the account can be used for remote logins through the rlogind or telnetd programs.

Tty Specifies the terminal of the originating activity. If this parameter is a null pointer or a null string, no tty origin checking is done.

Msg Returns an informative message indicating why the loginrestrictions subroutine failed. Upon return, the value is either a pointer to a valid string within memory allocated storage or a null value. If a message is displayed, it is provided based on the user interface.

Security

Access Control: The calling process must have access to the account information in the user database and the port information in the port database.

File Accessed:

ModeFilesr/etc/security/userr/etc/security/login.cfgr/etc/security/portlogr/etc/passwd

Return Values

If the account is valid for the specified usage, the **loginrestrictions** subroutine returns a value of 0. Otherwise, a value of -1 is returned, the **errno** global value is set to the appropriate error code, and the *Msg* parameter returns an informative message explaining why the specified account usage is invalid.

Error Codes

The **loginrestrictions** subroutine fails if one or more of the following values is true:

ENOENT The user specified does not have an account.

ESTALE The user's account is expired.

EPERM The user's account is locked, the specified terminal is locked, the user has had too many unsuccessful

login attempts, or the user cannot log in because the /etc/nologin file exists.

EACCES One of the following conditions exists:

· The specified terminal does not have access to the specified account.

- The Mode parameter is the S_SU value and the current process is not permitted to use the su
 command to access the specified user.
- · Access to the account is not permitted in the specified mode.
- · Access to the account is not permitted at the current time.
- · Access to the system with the specified terminal is not permitted at the current time.

EAGAIN The *Mode* parameter is either the **S LOGIN** value or the **S RLOGIN** value, and all the user licenses are

in the

in use

EINVAL The Mode parameter has a value other than S_LOGIN, S_SU, S_DAEMON, S_RLOGIN, or 0.

Related Information

The **authenticate** ("authenticate Subroutine" on page 94) subroutine, **getpcred** ("getpcred Subroutine" on page 341) subroutine, **getpenv** ("getpenv Subroutine" on page 343) subroutine, **loginfailed** ("loginfailed Subroutine" on page 636) subroutine, **loginsuccess** ("loginsuccess Subroutine" on page 641) subroutine, **setpcred** subroutine, **setpenv** subroutine.

The cron daemon.

The login command, rlogin command, telnet, tn, or tn3270 command, su command.

List of Security and Auditing Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Subroutines, Example Programs, and Libraries in *AIX 5L Version 5.2 General Programming Concepts:* Writing and Debugging Programs.

loginsuccess Subroutine

Purpose

Records a successful log in.

Library

Security Library (libc.a)

Syntax

```
#include <usersec.h>
int loginsuccess (User, Host, Tty, Msg)
char * User:
char * Host;
char * Tty;
char ** Msg;
```

Note: This subroutine is not thread-safe.

Description

The loginsuccess subroutine performs the processing necessary when a user successfully logs into the system. This subroutine updates the following attributes in the /etc/security/lastlog file for the specified user:

time_last_login Contains the current time.

tty_last_login Contains the value specified by the *Tty* parameter.

host_last_login Contains the value specified by the *Host* parameter or the local host

name if the *Host* parameter is a null value.

unsuccessful_login_count Indicates the number of unsuccessful login attempts. The

loginsuccess subroutine resets this attribute to a value of 0.

Additionally, a login success audit record is cut to indicate in the audit trail that this user has successfully logged in.

A message is returned in the Msg parameter that indicates the time, host, and port of the last successful and unsuccessful login. The number of unsuccessful login attempts since the last successful login is also provided to the user.

Parameters

User Specifies the login name of the user who has successfully logged in.

Specifies the name of the host from which the user logged in. If the Host parameter is a null value, the name Host of the local host is used.

Tty Specifies the name of the terminal which the user used to log in.

Returns a message indicating the delete time, host, and port of the last successful and unsuccessful logins. Msq The number of unsuccessful login attempts since the last successful login is also provided. Upon return, the value is either a pointer to a valid string within memory allocated storage or a null pointer. It is the responsibility of the calling program to free() the returned storage.

Security

Access Control: The calling process must have access to the account information in the user database.

File Accessed:

Mode File

/etc/security/lastlog rw

Auditing Events:

Event Information USER_Login username

Return Values

Upon successful completion, the loginsuccess subroutine returns a value of 0. Otherwise, a value of -1 is returned and the errno global value is set to indicate the error.

Error Codes

The loginsuccess subroutine fails if one or more of the following values is true:

ENOENT The specified user does not exist.

EACCES The current process does not have write access to the user database. **EPERM** The current process does not have permission to write an audit record.

Related Information

The authenticate ("authenticate Subroutine" on page 94) subroutine, getpcred ("getpcred Subroutine" on page 341) subroutine, getpenv ("getpenv Subroutine" on page 343) subroutine, loginfailed ("loginfailed Subroutine" on page 636) subroutine, loginrestrictions ("loginrestrictions Subroutine" on page 638) subroutine, **setpcred** subroutine, **setpenv** subroutine.

List of Security and Auditing Services in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Irint, Irintf, or Irintl Subroutine

Purpose

Rounds to nearest integer value using the current rounding direction.

```
#include <math.h>
long lrint (x)
double x;
long lrintf (x)
float x;
long lrintl (x)
long double x;
```

The **Irint**, **Irintf**, and **Irintl** subroutines round the x parameter to the nearest integer value, rounding according to the current rounding direction.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be rounded.

Return Values

Upon successful completion, the Irint, Irintf, and Irintl subroutines return the rounded integer value.

If x is NaN, a domain error occurs and an unspecified value is returned.

If x is +Inf, a domain error occurs and an unspecified value is returned.

If x is -Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a long, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a long, a domain error occurs and an unspecified value is returned.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "Ilrint, Ilrintf, or Ilrintl Subroutine" on page 616.

math.h in AIX 5L Version 5.2 Files Reference.

Iround, Iroundf, or Iroundl Subroutine

Purpose

Rounds to the nearest integer value.

```
#include <math.h>
long lround (x)
double x;
long lroundf (x)
float x;
long lroundl (x)
long double x;
```

The **Iround**, **Iroundf**, and **Iroundl** subroutines round the *x* parameter to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

x Specifies the value to be rounded.

Return Values

Upon successful completion, the **Iround**, **Iroundf**, and **Iroundl** subroutines return the rounded integer value.

If x is NaN, a domain error occurs and an unspecified value is returned.

If x is +Inf, a domain error occurs and an unspecified value is returned.

If x is -Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a **long**, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a **long**, a domain error occurs and an unspecified value is returned.

Related Information

"feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "Ilround, Ilroundf, or Ilroundl Subroutine" on page 617.

math.h in AIX 5L Version 5.2 Files Reference.

Isearch or Ifind Subroutine

Purpose

Performs a linear search and update.

Library

Standard C Library (libc.a)

```
void *lsearch (Key, Base, NumberOfElementsPointer, Width, ComparisonPointer)
const void *Key;
void *Base;
size_t Width, *NumberOfElementsPointer;
int (*ComparisonPointer) (cont void*, const void*);
```

```
void *lfind (Key, Base, NumberOfElementsPointer, Width, ComparisonPointer)
const void *Key, Base;
size t Width, *NumberOfElementsPointer;
int (*ComparisonPointer) (cont void*, const void*);
```

Warning: Undefined results can occur if there is not enough room in the table for the Isearch subroutine to add a new item.

The **Isearch** subroutine performs a linear search.

The algorithm returns a pointer to a table where data can be found. If the data is not in the table, the program adds it at the end of the table.

The Ifind subroutine is identical to the Isearch subroutine, except that if the data is not found, it is not added to the table. In this case, a NULL pointer is returned.

The pointers to the Key parameter and the element at the base of the table should be of type pointer-to-element and cast to type pointer-to-character. The value returned should be cast into type pointer-to-element.

The comparison function need not compare every byte; therefore, the elements can contain arbitrary data in addition to the values being compared.

Parameters

Base Points to the first element in the table.

ComparisonPointer Specifies the name (that you supply) of the comparison function

(strcmp, for example). It is called with two parameters that point to the

elements being compared.

Specifies the data to be sought in the table. Kev

NumberOfElementsPointer Points to an integer containing the current number of elements in the

table. This integer is incremented if the data is added to the table.

Width Specifies the size of an element in bytes.

The comparison function compares its parameters and returns a value as follows:

- If the first parameter equals the second parameter, the ComparisonPointer parameter returns a value of
- If the first parameter does not equal the second parameter, the ComparisonPointer parameter returns a value of 1.

Return Values

If the sought entry is found, both the **Isearch** and **Ifind** subroutines return a pointer to it. Otherwise, the Ifind subroutine returns a null pointer and the Isearch subroutine returns a pointer to the newly added element.

Related Information

The **bsearch** ("bsearch Subroutine" on page 102) subroutine, **hsearch** ("hsearch, hcreate, or hdestroy Subroutine" on page 425) subroutine, **qsort** subroutine, **tsearch** subroutine.

Donald E. Knuth. The Art of Computer Programming, Volume 3, 6.1, Algorithm S. Reading, Massachusetts: Addison-Wesley, 1981.

Searching and Sorting Example Program and Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Iseek, Ilseek or Iseek64 Subroutine

Purpose

Moves the read-write file pointer.

Library

Standard C Library (libc.a)

Syntax

```
off_t lseek ( FileDescriptor, Offset, Whence)
int FileDescriptor, Whence;
off_t Offset;
offset_t llseek (FileDescriptor, Offset, Whence)
int FileDescriptor, Whence;
offset_t Offset;
off64_t lseek64 (FileDescriptor, Offset, Whence)
int FileDescriptor, Whence;
off64 t Offset;
```

Description

The **Iseek**, **Ilseek**, and **Iseek64** subroutines set the read-write file pointer for the open file specified by the *FileDescriptor* parameter. The **Iseek** subroutine limits the *Offset* to **OFF_MAX**.

In the large file enabled programming environment, Iseek subroutine is redefined to Iseek64.

Parameters

FileDescriptor

Specifies a file descriptor obtained from a successful open or fcntl subroutine.

Offset Speci

Specifies a value, in bytes, that is used in conjunction with the Whence parameter to set the

file pointer. A negative value causes seeking in the reverse direction.

Whence

Specifies how to interpret the *Offset* parameter by setting the file pointer associated with the *FileDescriptor* parameter to one of the following variables:

SEEK SET

Sets the file pointer to the value of the *Offset* parameter.

SEEK_CUR

Sets the file pointer to its current location plus the value of the Offset parameter.

SEEK_END

Sets the file pointer to the size of the file plus the value of the *Offset* parameter.

Return Values

Upon successful completion, the resulting pointer location, measured in bytes from the beginning of the file, is returned. If either the **Iseek** or **Ilseek** subroutines are unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **Iseek** or **Ilseek** subroutines are unsuccessful and the file pointer remains unchanged if any of the following are true:

EBADF The FileDescriptor parameter is not an open file descriptor.

ESPIPE The FileDescriptor parameter is associated with a pipe (FIFO) or a socket.

The resulting offset would be greater than the maximum offset allowed for the file or device **EINVAL**

associated with FileDescriptor.

EOVERFLOW The resulting offset is larger than can be returned properly.

Files

/usr/include/unistd.h Defines standard macros, data types and subroutines.

Related Information

The fcntl ("fcntl, dup, or dup2 Subroutine" on page 214) subroutine, fseek, rewind, ftell, fgetpos, or fsetpos ("fseek, fseeko, fseeko64, rewind, ftell, ftello, ftello64, fgetpos, fgetpos64, fsetpos, or fsetpos64 Subroutine" on page 273) subroutine, open, openx, or creat ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine, read, ready, or readyx subroutine, write, writev, or writevx subroutine.

File Systems and Directories in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Ivm_querylv Subroutine

Purpose

Queries a logical volume and returns all pertinent information.

Library

Logical Volume Manager Library (liblvm.a)

Syntax

```
#include <1vm.h>
int lvm_querylv ( LV ID, QueryLV, PVName)
struct lv id *LV ID;
struct querylv **QueryLV;
char *PVName:
```

Description

Note: The lvm querylv subroutine uses the sysconfig system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the **lvm querylv** subroutine.

The **Ivm queryIv** subroutine returns information for the logical volume specified by the *LV ID* parameter.

The **querylv** structure, found in the **lvm.h** file, is defined as follows:

```
struct querylv {
      char lvname[LVM NAMESIZ];
      struct unique id vg id;
      long maxsize;
      long mirror policy;
      long lv state;
      long currentsize;
```

```
long ppsize;
      long permissions;
      long bb relocation;
      long write_verify;
      long mirwrt consist;
      long open close;
      struct pp *mirrors[LVM NUMCOPIES]
struct pp {
      struct unique_id pv_id;
      long lp num;
      long pp num;
      long ppstate;
```

Field

Description

Ivname

Specifies the special file name of the logical volume and can be either the full path name or a single file name that must reside in the /dev directory (for example, rhd1). All name fields must be null-terminated strings of from 1 to LVM_NAMESIZ bytes, including the null byte. If a raw or character device is not specified for the Ivname field, the Logical Volume Manager (LVM) will add an r to the file name to have a raw device name. If there is no raw device entry for this name, the LVM will return the LVM_NOTCHARDEV error code.

vg_id

Specifies the unique ID of the volume group that contains the logical volume.

maxsize

Indicates the maximum size in logical partitions for the logical volume and must be in the

range of 1 to LVM_MAXLPS.

mirror_policy

Specifies how the physical copies are written. The mirror policy field should be either LVM SEQUENTIAL or LVM PARALLEL to indicate how the physical copies of a logical partition are to be written when there is more than one copy.

lv_state

Specifies the current state of the logical volume and can have any of the following

bit-specific values ORed together:

LVM LVDEFINED

The logical volume is defined.

LVM_LVSTALE

The logical volume contains stale partitions.

currentsize

Indicates the current size in logical partitions of the logical volume. The size, in bytes, of

every physical partition is 2 to the power of the ppsize field.

ppsize permissions Specifies the size of the physical partitions of all physical volumes in the volume group. Specifies the permission assigned to the logical volume and can be one of the following values:

LVM_RDONLY

Access to this logical volume is read only.

LVM RDWR

Access to this logical volume is read/write.

bb_relocation

Specifies if bad block relocation is desired and is one of the following values:

LVM NORELOC

Bad blocks will not be relocated.

LVM RELOC

Bad blocks will be relocated.

write_verify

Specifies if write verification for the logical volume is desired and returns one of the following values:

LVM NOVERIFY

Write verification is not performed for this logical volume.

LVM_VERIFY

Write verification is performed on all writes to the logical volume.

Field mirwrt_consist

open_close

mirrors

Description

Indicates whether mirror-write consistency recovery will be performed for this logical volume.

The LVM always ensures data consistency among mirrored copies of a logical volume during normal I/O processing. For every write to a logical volume, the LVM generates a write request for every mirror copy. A problem arises if the system crashes in the middle of processing a mirrored write (before all copies are written). If mirror write consistency recovery is requested for a logical volume, the LVM keeps additional information to allow recovery of these inconsistent mirrors. Mirror write consistency recovery should be performed for most mirrored logical volumes. Logical volumes, such as page space, that do not use the existing data when the volume group is re-varied on do not need this protection.

Values for the **mirwrt consist** field are:

LVM CONSIST

Mirror-write consistency recovery will be done for this logical volume.

Mirror-write consistency recovery will not be done for this logical volume. Specifies if the logical volume is opened or closed. Values for this field are:

LVM QLV NOTOPEN

The logical volume is closed.

LVM QLVOPEN

The logical volume is opened by one or more processes.

Specifies an array of pointers to partition map lists (physical volume id, logical partition number, physical partition number, and physical partition state for each copy of the logical

partitions for the logical volume). The ppstate field can be LVM_PPFREE,

LVM_PPALLOC, or LVM_PPSTALE. If a logical partition does not contain any copies, its

pv_id, lp_num, and pp_num fields will contain zeros.

The PVName parameter enables the user to query from a volume group descriptor area on a specific physical volume instead of from the Logical Volume Manager's (LVM) most recent, in-memory copy of the descriptor area. This method should only be used if the volume group is varied off.

Note: The data returned is not guaranteed to be the most recent or correct, and it can reflect a back-level descriptor area.

The *PVName* parameter should specify either the full path name of the physical volume that contains the descriptor area to guery, or a single file name that must reside in the /dev directory (for example, rhdisk1). This parameter must be a null-terminated string between 1 and LVM NAMESIZ bytes, including the null byte, and must represent a raw device entry. If a raw or character device is not specified for the PVName parameter, the LVM adds an r to the file name to have a raw device name. If there is no raw device entry for this name, the LVM returns the LVM NOTCHARDEV error code.

If a *PVName* parameter is specified, only the **minor num** field of the *LV ID* parameter need be supplied. The LVM fills in the vg_id field and returns it to the user. If the user wishes to query from the LVM's in-memory copy, the PVName parameter should be set to null. When using this method of query, the volume group must be varied on, or an error is returned.

Note: As long as the PVName parameter is not null, the LVM will attempt a query from a physical volume and not from its in-memory copy of data.

In addition to the PVName parameter, the caller passes the ID of the logical volume to be gueried (LV ID parameter) and the address of a pointer to the queryly structure, specified by the QueryLV parameter. The LVM separately allocates the space needed for the queryly structure and the struct pp arrays, and

returns the querylv structure's address in the pointer variable passed in by the user. The user is responsible for freeing the space by first freeing the struct pp pointers in the mirrors array and then freeing the querylv structure.

Parameters

 $LV_{\perp}ID$ Points to an Iv_id structure that specifies the logical volume to query.

QuervLV Contains the address of a pointer to the querylv structure.

PVName Names the physical volume from which to use the volume group descriptor for the query. This

parameter can also be null.

Return Values

If the **lvm_querylv** subroutine is successful, it returns a value of 0.

Error Codes

If the Ivm_queryIv subroutine does not complete successfully, it returns one of the following values:

The subroutine could not allocate enough space for the complete buffer. LVM ALLOCERR

LVM_INVALID_MIN_NUM The minor number of the logical volume is not valid. LVM INVALID PARAM A parameter passed into the routine is not valid.

LVM INV DEVENT The device entry for the physical volume specified by the *Pvname* parameter is

not valid and cannot be checked to determine if it is raw.

LVM_NOTCHARDEV The physical volume name given does not represent a raw or character device.

LVM_OFFLINE The volume group containing the logical volume to guery was offline.

If the query originates from the varied-on volume group's current volume group

descriptor area, one of the following error codes is returned:

LVM_DALVOPN The volume group reserved logical volume could not be opened.

The volume group is currently locked because system management on the LVM_MAPFBSY

volume group is being done by another process.

The mapped file, which contains a copy of the volume group descriptor area LVM_MAPFOPN

used for making changes to the volume group, could not be opened.

LVM_MAPFRDWR The mapped file could not be read or written.

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, one of the following error codes is returned:

LVM BADBBDIR The bad-block directory could not be read or written.

The LVM record, which contains information about the volume group descriptor area, could LVM_LVMRECERR

not be read.

There are no volume group descriptor areas on the physical volume specified. LVM_NOPVVGDA

LVM_NOTVGMEM The physical volume specified is not a member of a volume group.

LVM_PVDAREAD An error occurred while trying to read the volume group descriptor area from the specified

physical volume.

LVM_PVOPNERR The physical volume device could not be opened.

A bad block was found in the volume group descriptor area located on the physical volume LVM VGDA BB

that was specified for the query. Therefore, a query cannot be done from the specified

physical volume.

Related Information

List of Logical Volume Subroutines and Logical Volume Programming Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Ivm_querypv Subroutine

Purpose

Queries a physical volume and returns all pertinent information.

Library

Logical Volume Manager Library (liblvm.a)

Syntax

```
#include <1vm.h>
int lvm_querypv (VG_ID, PV_ID, QueryPV, PVName)
struct unique_id * VG_ID;
struct unique_id * PV_ID;
struct querypv ** QueryPV;
char * PVName;
```

Description

Note: The **lvm_querypv** subroutine uses the **sysconfig** system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the **lvm_querypv** subroutine.

The **lvm_querypv** subroutine returns information on the physical volume specified by the *PV_ID* parameter.

The **querypv** structure, defined in the **lvm.h** file, contains the following fields:

```
struct querypv {
      long ppsize;
      long pv_state;
      long pp_count;
      long alloc ppcount;
      struct pp map *pp map;
      long pvnum vgdas;
struct pp_map {
      long pp state;
      struct lv id lv id;
      long lp num;
      long copy;
      struct unique id fst_alt_vol;
      long fst alt part;
      struct unique id snd alt vol;
      long snd alt part;
 }
```

FieldDescriptionppsizeSpecifies the size of the physical partitions, which is the same for all partitions within a
volume group. The size in bytes of a physical partition is 2 to the power of ppsize.pv_stateContains the current state of the physical volume.

pp_count Contains the total number of physical partitions on the physical volume.

alloc ppcount Contains the number of allocated physical partitions on the physical volume.

Field

pp map

Description

Points to an array that has entries for each physical partition of the physical volume. Each entry in this array will contain the pp state that specifies the state of the physical partition (LVM_PPFREE, LVM_PPALLOC, or LVM_PPSTALE) and the lv id, field, the ID of the logical volume that it is a member of. The pp_map array also contains the physical volume IDs (fst alt vol and snd alt vol) and the physical partition numbers (fst alt part and snd alt part) for the first and second alternate copies of the physical partition, and the logical partition number (1p num) that the physical partition corresponds to.

If the physical partition is free (that is, not allocated), all of its pp_map fields will be zero.

fst alt vol

Contains zeros if the logical partition has only one physical copy.

fst_alt_part

Contains zeros if the logical partition has only one physical copy.

snd_alt_vol

Contains zeros if the logical partition has only one or two physical copies.

snd alt part

Contains zeros if the logical partition has only one or two physical copies.

Specifies which copy of a logical partition this physical partition is allocated to. сору This field will contain one of the following values:

LVM_PRIMARY

Primary and only copy of a logical partition

LVM PRIMOF2

Primary copy of a logical partition with two physical copies

LVM PRIMOF3

Primary copy of a logical partition with three physical copies

LVM_SCNDOF2

Secondary copy of a logical partition with two physical copies

LVM SCNDOF3

Secondary copy of a logical partition with three physical copies

LVM_TERTOF3

Tertiary copy of a logical partition with three physical copies.

pvnum_vgdas

Contains the number of volume group descriptor areas (0, 1, or 2) that are on the specified physical volume.

The *PVName* parameter enables the user to guery from a volume group descriptor area on a specific physical volume instead of from the Logical Volume Manager's (LVM) most recent, in-memory copy of the descriptor area. This method should only be used if the volume group is varied off. The data returned is not guaranteed to be most recent or correct, and it can reflect a back level descriptor area.

The PVname parameter should specify either the full path name of the physical volume that contains the descriptor area to guery or a single file name that must reside in the /dev directory (for example, rhdisk1). This field must be a null-terminated string of from 1 to LVM NAMESIZ bytes, including the null byte, and represent a raw or character device. If a raw or character device is not specified for the PVName parameter, the LVM will add an r to the file name in order to have a raw device name. If there is no raw device entry for this name, the LVM will return the LVM NOTCHARDEV error code. If a PVName is specified, the volume group identifier, VG ID, will be returned by the LVM through the VG ID parameter passed in by the user. If the user wishes to query from the LVM in-memory copy, the PVName parameter should be set to null. When using this method of query, the volume group must be varied on, or an error will be returned.

Note: As long as the *PVName* is not null, the LVM will attempt a guery from a physical volume and *not* from its in-memory copy of data.

In addition to the *PVName* parameter, the caller passes the *VG_ID* parameter, indicating the volume group that contains the physical volume to be queried, the unique ID of the physical volume to be queried, the PV ID parameter, and the address of a pointer of the type QueryPV. The LVM will separately allocate enough space for the querypv structure and the struct pp map array and return the address of the querypv structure in the QueryPV pointer passed in. The user is responsible for freeing the space by freeing the struct pp_map pointer and then freeing the QueryPV pointer.

Parameters

VG ID Points to a unique_id structure that specifies the volume group of which the physical volume to query

PV_ID Points to a unique_id structure that specifies the physical volume to query.

QueryPV Specifies the address of a pointer to a querypv structure.

PVName Names a physical volume from which to use the volume group descriptor area for the query. This

parameter can be null.

Return Values

The **lvm_querypv** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm querypv** subroutine fails it returns one of the following error codes:

LVM_ALLOCERR The routine cannot allocate enough space for a complete buffer.

LVM_INVALID_PARAM An invalid parameter was passed into the routine.

LVM_INV_DEVENT The device entry for the physical volume is invalid and cannot be checked to

determine if it is raw.

LVM OFFLINE The volume group specified is offline and should be online.

If the query originates from the varied-on volume group's current volume group descriptor area, one of the following error codes may be returned:

LVM DALVOPN The volume group reserved logical volume could not be opened.

The volume group is currently locked because system management on the volume group is LVM_MAPFBSY

being done by another process.

The mapped file, which contains a copy of the volume group descriptor area used for making LVM_MAPFOPN

changes to the volume group, could not be opened.

Either the mapped file could not be read, or it could not be written. LVM MAPFRDWR

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, then one of the following error codes may be returned:

LVM_BADBBDIR The bad-block directory could not be read or written.

LVM_LVMRECERR The LVM record, which contains information about the volume group descriptor area,

could not be read.

LVM_NOPVVGDA There are no volume group descriptor areas on this physical volume.

LVM NOTCHARDEV A device is not a raw or character device.

LVM NOTVGMEM The physical volume is not a member of a volume group.

LVM_PVDAREAD An error occurred while trying to read the volume group descriptor area from the specified

physical volume.

LVM PVOPNERR The physical volume device could not be opened. A bad block was found in the volume group descriptor area located on the physical volume that was specified for the query. Therefore, a query cannot be done from the specified physical volume.

Related Information

List of Logical Volume Subroutines and Logical Volume Programming Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Ivm_queryvg Subroutine

Purpose

Queries a volume group and returns pertinent information.

Library

Logical Volume Manager Library (liblvm.a)

Syntax

```
#include <1vm.h>
int lvm_queryvg ( VG_ID, QueryVG, PVName)
struct unique_id *VG_ID;
struct queryvg **QueryVG;
char *PVName;
```

Description

Note: The **lvm_queryvg** subroutine uses the **sysconfig** system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the **lvm_queryvg** subroutine.

The **lvm_queryvg** subroutine returns information on the volume group specified by the *VG_ID* parameter.

The **queryvg** structure, found in the **lvm.h** file, contains the following fields:

```
struct queryvg {
     long maxlvs;
     long ppsize;
     long freespace;
     long num lvs;
     long num_pvs;
     long total vgdas;
     struct lv array *lvs;
     struct pv_array *pvs;
struct pv_array {
    struct unique id pv id;
     long pvnum vgdas;
     char state;
     char res[3];
 }
struct lv array {
     struct lv id
                    lv id;
     char lvname[LVM_NAMESIZ];
     char state;
     char res[3];
```

Description Field

Specifies the maximum number of logical volumes allowed in the volume group. maxlvs Specifies the size of all physical partitions in the volume group. The size in bytes of ppsize

each physical partitions is 2 to the power of the ppsize field.

freespace Contains the number of free physical partitions in this volume group.

num lvs Indicates the number of logical volumes. num pvs Indicates the number of physical volumes.

Specifies the total number of volume group descriptor areas for the entire volume total vgdas

1vs Points to an array of unique IDs, names, and states of the logical volumes in the

volume group.

Points to an array of unique IDs, states, and the number of volume group descriptor pvs

areas for each of the physical volumes in the volume group.

The PVName parameter enables the user to query from a descriptor area on a specific physical volume instead of from the Logical Volume Manager's (LVM) most recent, in-memory copy of the descriptor area. This method should only be used if the volume group is varied off. The data returned is *not guaranteed* to be most recent or correct, and it can reflect a back level descriptor area. The Pvname parameter should specify either the full path name of the physical volume that contains the descriptor area to query or a single file name that must reside in the /dev directory (for example, rhdisk1). The name must represent a raw device. If a raw or character device is not specified for the PVName parameter, the Logical Volume Manager will add an r to the file name in order to have a raw device name. If there is no raw device entry for this name, the LVM returns the LVM_NOTCHARDEV error code. This field must be a null-terminated string of from 1 to LVM NAMESIZ bytes, including the null byte. If a PVName is specified, the LVM will return the VG ID to the user through the VG ID pointer passed in. If the user wishes to query from the LVM in-memory copy, the PVName parameter should be set to null. When using this method of query, the volume group must be varied on, or an error will be returned.

Note: As long as the PVName parameter is not null, the LVM will attempt a guery from a physical volume and not its in-memory copy of data.

In addition to the PVName parameter, the caller passes the unique ID of the volume group to be queried (VG_ID) and the address of a pointer to a queryvg structure. The LVM will separately allocate enough space for the queryvg structure, as well as the Iv_array and pv_array structures, and return the address of the completed structure in the QueryVG parameter passed in by the user. The user is responsible for freeing the space by freeing the 1v and pv pointers and then freeing the QueryVG pointer.

Parameters

 VG_ID Points to a unique_id structure that specifies the volume group to be queried.

QueryVG Specifies the address of a pointer to the **quervvq** structure.

PVName Specifies the name of the physical volume that contains the descriptor area to query and must

be the name of a raw device.

Return Values

The **lvm_queryvg**n subroutine returns a value of 0 upon successful completion.

Error Codes

LVM ALLOCERR

If the **lvm_queryvg** subroutine fails it returns one of the following error codes:

LVM_FORCEOFF The volume group has been forcefully varied off due to a loss of

quorum.

The subroutine cannot allocate enough space for a complete buffer.

An invalid parameter was passed into the routine. The volume group is offline and should be online.

If the query originates from the varied-on volume group's current volume group descriptor area, one of the following error codes may be returned:

LVM_DALVOPNThe volume group reserved logical volume could not be opened. **LVM_INV_DEVENT**The device entry for the physical volume specified by the *PVName*

parameter is invalid and cannot be checked to determine if it is raw.

LVM_MAPFBSY The volume group is currently locked because system management on the

volume group is being done by another process.

LVM_MAPFOPN The mapped file, which contains a copy of the volume group descriptor area

used for making changes to the volume group, could not be opened.

LVM_MAPFRDWR Either the mapped file could not be read, or it could not be written.

LVM NOTCHARDEV A device is not a raw or character device.

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, one of the following error codes may be returned:

LVM_BADBBDIR The bad-block directory could not be read or written.

LVM_LVMRECERR The LVM record, which contains information about the volume group

descriptor area, could not be read.

LVM_NOPVVGDAThere are no volume group descriptor areas on this physical volume.

LVM_NOTVGMEMThe physical volume is not a member of a volume group.

LVM_PVDAREAD An error occurred while trying to read the volume group descriptor area from

the specified physical volume.

LVM_PVOPNERR The physical volume device could not be opened.

LVM_VGDA_BB A bad block was found in the volume group descriptor area located on the

physical volume that was specified for the query. Therefore, a query cannot

be done from this physical volume.

Related Information

List of Logical Volume Subroutines and Logical Volume Programming Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Ivm_queryvgs Subroutine

Purpose

Queries volume groups and returns information to online volume groups.

Library

Logical Volume Manager Library (liblvm.a)

Syntax

#include <1vm.h>

int lvm_queryvgs (QueryVGS, Kmid)
struct queryvgs **QueryVGS;
mid_t Kmid;

Note: The **Ivm queryvgs** subroutine uses the **sysconfig** system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the **lvm queryvgs** subroutine.

The **lvm queryvgs** subroutine returns the volume group IDs and major numbers for all volume groups in the system that are online.

The caller passes the address of a pointer to a queryvgs structure, and the Logical Volume Manager (LVM) allocates enough space for the structure and returns the address of the structure in the pointer passed in by the user. The caller also passes in a Kmid parameter, which identifies the entry point of the logical device driver module:

```
struct queryvgs {
       long num vgs;
       struct {
       long major num
       struct unique id vg id;
       } vgs [LVM MAXVGS];
}
```

Field

Description

num vgs

Contains the number of online volume groups on the system. The vgs is an array of the volume group IDs and major numbers of all online volume groups in the system.

Parameters

QuervVGS

Points to the **queryvgs** structure.

Kmid

Identifies the address of the entry point of the logical volume device driver module.

Return Values

The **lvm_queryvgs** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm queryvgs** subroutine fails, it returns one of the following error codes:

LVM_ALLOCERR LVM_INVALID_PARAM The routine cannot allocate enough space for the complete buffer.

An invalid parameter was passed into the routine.

LVM_INVCONFIG

An error occurred while attempting to configure this volume group into the kernel. This error will normally result if the module ID is invalid, if the major number given

is already in use, or if the volume group device could not be opened.

Related Information

List of Logical Volume Subroutines and Logical Volume Programming Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, or valloc Subroutine

Purpose

Provides a memory allocator.

Libraries

Berkeley Compatibility Library (libbsd.a)

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
void *malloc (Size)
size t Size;
void free (Pointer)
void *Pointer;
void *realloc (Pointer, Size)
void *Pointer;
size_t Size;
void *calloc (NumberOfElements, ElementSize)
size t NumberOfElements;
size_t ElementSize;
char *alloca (Size)
int Size;
void *valloc (Size)
size t Size;
#include <malloc.h>
#include <stdlib.h>
int mallopt (Command, Value)
int Command;
int Value:
struct mallinfo mallinfo( )
struct mallinfo heap mallinfo heap (Heap)
int Heap;
```

Description

The **malloc** and **free** subroutines provide a general-purpose memory allocation package.

The **malloc** subroutine returns a pointer to a block of memory of at least the number of bytes specified by the Size parameter. The block is aligned so that it can be used for any type of data. Undefined results occur if the space assigned by the **malloc** subroutine is overrun.

The free subroutine frees a block of memory previously allocated by the malloc subroutine. Undefined results occur if the *Pointer* parameter is not a valid pointer. If the *Pointer* parameter is a null value, no action will occur.

The realloc subroutine changes the size of the block of memory pointed to by the Pointer parameter to the number of bytes specified by the Size parameter and returns a new pointer to the block. The pointer specified by the *Pointer* parameter must have been created with the malloc, calloc, or realloc subroutines and not been deallocated with the **free** or **realloc** subroutines. If the *Pointer* parameter is NULL, the realloc subroutine calls the malloc subroutine, creating a new block of memory. Otherwise, undefined results occur when the *Pointer* parameter does not point to a currently allocated block of memory.

The contents of the block returned by the realloc subroutine remain unchanged up to the lesser of the old and new sizes. If a large enough block of memory is not available, the realloc subroutine acquires a new

area and moves the data to the new space. The *Pointer* parameter remains unchanged. Because of this, always use the return value to avoid memory corruption. The realloc subroutine supports the old realloc protocol wherein the realloc protocol returns a pointer to a previously freed block of memory if that block satisfies the realloc request. The realloc subroutine searches a list, maintained by the free subroutine, of the ten most recently freed blocks of memory. If the list does not contain a memory block that satisfies the specified Size parameter, the realloc subroutine calls the malloc subroutine. This list is cleared by calls to the malloc, calloc, valloc, or realloc subroutines.

The calloc subroutine allocates space for an array with the number of elements specified by the NumberOfElements parameter. The ElementSize parameter specifies in bytes each element, and initializes space to zeros. The order and contiquity of storage allocated by successive calls to the calloc subroutine is unspecified. The pointer returned points to the first (lowest) byte address of the allocated space.

The valloc subroutine, found in many BSD systems, is supported as a compatibility interface in the Berkeley Compatibility Library (libbsd.a). The valloc subroutine calls the malloc subroutine and automatically page-aligns requests that are greater than one page. The only difference between the valloc subroutine in the libbsd.a library and the one in the standard C library (described above) is in the value returned when the size parameter is zero.

The **alloca** subroutine allocates the number of bytes of space specified by the *Size* parameter in the stack frame of the caller. This space is automatically freed when the subroutine that called the alloca subroutine returns to its caller.

If alloca is used in the code and compiled with the C++ compiler, #pragma alloca would also have to be added before the usage of alloca in the code. Alternatively, the tag -ma would have to be used while compiling the code.

The valloc subroutine has the same effect as malloc, except that the allocated memory is aligned to a multiple of the value returned by **sysconf**(SC PAGESIZE).

The mallopt and mallinfo subroutines are provided for source-level compatibility with the System V malloc subroutine. Nothing done with the mallopt subroutine affects how memory is allocated by the system, unless the M MXFAST option is used...

The mallinfo subroutine can be used to obtain information about the heap managed by the malloc subroutine. Refer to the malloc.h file for details of the mallinfo structure.

Note: When MALLOCTYPE is set to buckets and the memory request is within the range of block sizes defined for the buckets, the memory request is serviced but the heap statistics that are reported by mallinfo are not updated.

The mallinfo_heap subroutine provides information about a specific heap if MULTIHEAPS is enabled. The mallinfo_heap subroutine returns a structure that details the properties and statistics of the heap specified by the user. Refer to the malloc.h file for details about the mallinfo heap structure.

Note: The **mallinfo heap** subroutine should not be used with Malloc 3.1.

Note: AIX uses a delayed paging slot allocation technique for storage allocated to applications. When storage is allocated to an application with a subroutine such as malloc, no paging space is assigned to that storage until the storage is referenced. This technique is useful for applications that allocate large sparse memory segments. However, this technique may affect portability of applications that allocate very large amounts of memory. If the application expects that calls to malloc will fail when there is not enough backing storage to support the memory request, the application may allocate too much memory. When this memory is referenced later, the machine quickly runs out of paging space and the operating system kills processes so that the system is not completely exhausted of virtual memory. The application that allocates memory must ensure that

backing storage exists for the storage being allocated. Setting the PSALLOC environment variable to PSALLOC=early changes the paging space allocation technique to an early allocation algorithm. In early allocation, paging space is assigned once the memory is requested. See the Paging Space and Virtual Memory in the AIX 5L Version 5.2 System Management Concepts: Operating System and Devices for more information.

Parameters

Size Specifies a number of bytes of memory.

Pointer Points to the block of memory that was returned by the malloc or calloc subroutines.

The Pointer parameter points to the first (lowest) byte address of the block.

Command Specifies a mallopt subroutine command. If M DISCLAIM is used, then the paging

> space and physical memory in use by freed malloc space is returned to the system resource pool. If they are needed to fulfill a malloc request, they will be allocated to the process as needed. The address space is not altered. This will only release whole pages at a time. The M MXFAST command can change how the system allocates memory by enabling or disabling the default and 3.1 allocator. The M_NLBLKS, M_GRAIN, and M_KEEP commands are only provided for source code compatability,

and do not provide any functionality.

Specifies the value to which the M_DISCLAIM, M_MXFAST, M_NLBLKS, M_GRAIN, Value

> or M_KEEP label is to be set. M_NLBLKS, M_GRAIN, and M_KEEP are provided only for source code compatibility. They do not affect the operation of subsequent calls to the malloc subroutine. For M MXFAST, setting Value to 0 disables the 3.1 allocator, and enable the default allocator. Setting Value to a positive value will disable the default allocator, and enable the 3.1 allocator. For M_DISCLAIM, Value

does not affect the behavior of mallopt(), and should be set to NULL.

NumberOfElements Specifies the number of elements in the array. **ElementSize** Specifies the size of each element in the array.

Heap Specifies a heap number from 0 to 31.

Return Values

Each of the allocation subroutines returns a pointer to space suitably aligned for storage of any type of object. Cast the pointer to the pointer-to-element type before using it.

The malloc, realloc, calloc, and valloc subroutines return a null pointer if there is no available memory, or if the memory arena has been corrupted by being stored outside the bounds of a block. When this happens, the block pointed to by the *Pointer* parameter may be destroyed.

If the malloc or valloc subroutine is called with a size of 0, the subroutine returns a null pointer. If the realloc subroutine is called with a nonnull pointer and a size of 0, the realloc subroutine attempts to free the pointer and return a null pointer. If the **realloc** subroutine is called with a null pointer, it calls the malloc subroutine for the specified size and returns a non-null pointer if malloc succeeds or a null pointer if malloc fails.

Error Codes

When the memory allocation subroutines are unsuccessful, the global variable errno may be set to the following values:

EINVAL Indicates a call has requested 0 bytes.

ENOMEM Indicates that not enough storage space was available.

Related Information

The _end, _etext, or _edata ("_end, _etext, or _edata Identifier" on page 184) identifier.

User Defined Malloc Replacement, Debug Malloc, Malloc Multiheap, Malloc Buckets, Malloc Log, Malloc Trace, System Memory Allocation Using the malloc Subsystem, Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs, and Understanding Paging Space Allocation Policies section in AIX 5L Version 5.2 System Management Concepts: Operating System and Devices.

madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout, omout, fmout, m_out, sdiv, or itom **Subroutine**

Purpose

Multiple-precision integer arithmetic.

Library

Berkeley Compatibility Library (libbsd.a)

```
#include <mp.h>
#include <stdio.h>
typedef struct mint {int Length; short * Value} MINT;
madd(a, b, c)
msub(a,b,c)
\mathbf{mult}(a,b,c)
mdiv(a,b,q,r)
pow(a,b, m,c)
gcd(a,b,c)
invert(a,b,c)
rpow(a,n,c)
msqrt(a,b,r)
mcmp(a,b)
move(a,b)
min(a)
omin(a)
fmin(a,f)
m in(a, n, f)
mout(a)
omout(a)
fmout (a, f)
m_out(a,n,f)
MINT *a, *b, *c, *m, *q, *r;
FILE * f;
int n;
sdiv(a,n,q,r)
MINT *a, *q;
short n;
short *r;
MINT *itom(n)
```

These subroutines perform arithmetic on integers of arbitrary *Length*. The integers are stored using the defined type MINT. Pointers to a MINT can be initialized using the itom subroutine, which sets the initial *Value* to *n*. After that, space is managed automatically by the subroutines.

The **madd** subroutine, **msub** subroutine, and **mult** subroutine assign to c the sum, difference, and product, respectively, of a and b.

The **mdiv** subroutine assigns to q and r the quotient and remainder obtained from dividing a by b.

The **sdiv** subroutine is like the **mdiv** subroutine except that the divisor is a short integer n and the remainder is placed in a short whose address is given as r.

The **msqrt** subroutine produces the integer square root of a in b and places the remainder in r.

The **rpow** subroutine calculates in c the value of a raised to the (regular integral) power n, while the **pow** subroutine calculates this with a full multiple precision exponent b and the result is reduced modulo m.

Note: The pow subroutine is also present in the IEEE Math Library, libm.a, and the System V Math Library, libmsaa.a. The pow subroutine in libm.a or libmsaa.a may be loaded in error unless the libbsd.a library is listed before the libm.a or libmsaa.a library on the command line.

The **gcd** subroutine returns the greatest common denominator of a and b in c, and the **invert** subroutine computes c such that $a^*c \mod b=1$, for a and b relatively prime.

The **mcmp** subroutine returns a negative, 0, or positive integer value when a is less than, equal to, or greater than b. respectively.

The **move** subroutine copies a to b. The **min** subroutine and **mout** subroutine do decimal input and output while the **omin** subroutine and **omout** subroutine do octal input and output. More generally, the **fmin** subroutine and **fmout** subroutine do decimal input and output using file f, and the m_in subroutine and m_out subroutine do inputs and outputs with arbitrary radix n. On input, records should have the form of strings of digits terminated by a new line; output records have a similar form.

Programs that use the multiple-precision arithmetic functions must link with the libbsd.a library.

Bases for input and output should be less than or equal to 10.

pow is also the name of a standard math library routine.

Parameters

Lengtn	Specifies the length of an integer.
Value	Specifies the initial value to be used in the routine.
а	Specifies the first operand of the multiple-precision routines.
b	Specifies the second operand of the multiple-precision routines.
С	Contains the integer result.
f	A pointer of the type FILE that points to input and output files used with input/output routines.
m	Indicates modulo.
n	Provides a value used to specify radix with m_in and m_out , power with rpow , and divisor with sdiv .
q	Contains the quotient obtained from mdiv .
r	Contains the remainder obtained from mdiv, sdiv, and msqrt.

Error Codes

Error messages and core images are displayed as a result of illegal operations and running out of memory.

Files

/usr/lib/libbsd.a

Object code library.

Related Information

The **bc** command. **dc** command.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

madvise Subroutine

Purpose

Advises the system of expected paging behavior.

Library

Standard C Library (libc.a).

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>
int madvise( addr, len, behav)
caddr_t addr;
size t len;
int behav;
```

Description

The madvise subroutine permits a process to advise the system about its expected future behavior in referencing a mapped file region or anonymous memory region.

The **madvise** subroutine has no functionality and is supported for compatibility only.

Parameters

len

addr Specifies the starting address of the memory region. Must be a multiple of the page size returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter.

Specifies the length, in bytes, of the memory region. If the len value is not a multiple of page size as returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter, the length of the region will be rounded up to the next multiple of the page size.

behav

Specifies the future behavior of the memory region. The following values for the *behav* parameter are defined in the */usr/include/sys/mman.h* file:

Value Paging Behavior Message

MADV NORMAL

The system provides no further special treatment for the memory region.

MADV_RANDOM

The system expects random page references to that memory region.

MADV_SEQUENTIAL

The system expects sequential page references to that memory region.

MADV WILLNEED

The system expects the process will need these pages.

MADV DONTNEED

The system expects the process does not need these pages.

MADV SPACEAVAIL

The system will ensure that memory resources are reserved.

Return Values

When successful, the **madvise** subroutine returns 0. Otherwise, it returns -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **madvise** subroutine is unsuccessful, the **errno** global variable can be set to one of the following values:

EINVAL The *behav* parameter is invalid.

ENOSPC The behav parameter specifies MADV_SPACEAVAIL and resources cannot be reserved.

Related Information

The **mmap** ("mmap or mmap64 Subroutine" on page 690) subroutine, **sysconf** subroutine.

List of Memory Manipulation Services and Understanding Paging Space Programming Requirements in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

makecontext or swapcontext Subroutine

Purpose

Modifies the context specified by ucp.

Library

(libc.a)

Syntax

#include <ucontext.h>

```
void makecontext (ucontext_t *ucp, (void *func) (), int argc, ...); int swapcontext (uncontext t *oucp, const uncontext t *ucp);
```

The **makecontext** subroutine modifies the context specified by *ucp*, which has been initialized using getcontext subroutine. When this context is resumed using swapcontext subroutine or setcontext subroutine, program execution continues by calling *func* parameter, passing it the arguments that follow argc in the makecontext subroutine.

Before a call is made to makecontext subroutine, the context being modified should have a stack allocated for it. The value of argc must match the number of integer argument passed to func parameter, otherwise the behavior is undefined.

The uc link member is used to determine the context that will be resumed when the context being modified by makecontext subroutine returns. The uc_link member should be initialized prior to the call to makecontext subroutine.

The swapcontext subroutine function saves the current context in the context structure pointed to by oucp parameter and sets the context to the context structure pointed to by ucp.

Parameters

A pointer to a user structure. иср oucp A pointer to a user structure.

A pointer to a function to be called when *ucp* is restored. func The number of arguments being passed to *func* parameter. argc

Return Values

On successful completion, swapcontext subroutine returns 0. Otherwise, a value of -1 is returned and errno is set to indicate the error.

-1 Not successful and the errno global variable is set to one of the following error codes.

Error Codes

ENOMEM The *ucp* argument does not have enough stack left to complete the operation.

Related Information

The **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutine, exit ("exit, atexit, exit, or Exit Subroutine" on page 203) subroutine, wait subroutine, getcontext ("getcontext or setcontext Subroutine" on page 307) subroutine, sigaction subroutine, and sigprocmask subroutine.

matherr Subroutine

Purpose

Math error handling function.

Library

System V Math Library (libmsaa.a)

Syntax

#include <math.h>

```
int matherr (x)
struct exception *x;
```

The matherr subroutine is called by math library routines when errors are detected.

You can use **matherr** or define your own procedure for handling errors by creating a function named matherr in your program. Such a user-designed function must follow the same syntax as **matherr**. When an error occurs, a pointer to the exception structure will be passed to the user-supplied matherr function. This structure, which is defined in the **math.h** file, includes:

```
int type;
char *name;
double arg1, arg2, retval;
```

Parameters

type

Specifies an integer describing the type of error that has occurred from the following list defined by the **math.h** file:

DOMAIN

Argument domain error

SING Argument singularity

OVERFLOW

Overflow range error

UNDERFLOW

Underflow range error

TLOSS Total loss of significance

PLOSS

Partial loss of significance.

name Points to a string containing the name of the routine that caused the error.

arg1 Points to the first argument with which the routine was invoked.arg2 Points to the second argument with which the routine was invoked.

retval Specifies the default value that is returned by the routine unless the user's matherr function sets it to a

different value.

Return Values

If the user's **matherr** function returns a non-zero value, no error message is printed, and the **errno** global variable will not be set.

Error Codes

If the function **matherr** is not supplied by the user, the default error-handling procedures, described with the math library routines involved, will be invoked upon error. In every case, the **errno** global variable is set to **EDOM** or **ERANGE** and the program continues.

Related Information

The **bessel: j0**, **j1**, **jn**, **y0**, **y1**, **yn** ("bessel: **j0**, **j1**, **jn**, **y0**, **y1**, or yn Subroutine" on page 98) subroutine, **exp**, **expm1**, **log**, **log10**, **log1p**, **pow** ("exp, expf, or expl Subroutine" on page 205) subroutine, **lgamma** ("gamma Subroutine" on page 291) subroutine, **hypot**, **cabs** ("hypot, hypotf, or hypotl Subroutine" on page 426) subroutine, **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, **atan2** subroutine, **sinh**, **cosh**, **tanh** subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

MatchAllAuths, , MatchAnyAuths, or MatchAnyAuthsList Subroutine

Purpose

Compare authorizations.

Library

Security Library (libc.a)

Syntax

#include <usersec.h> int MatchAllAuths(CommaListOfAuths) char *CommaListOfAuths; int MatchAllAuthsList(CommaListOfAuths, NSListOfAuths) char *CommaListOfAuths; char *NSListOfAuths; int MatchAnyAuths(CommaListOfAuths) char *CommaListOfAuths; int MatchAnyAuthsList(CommaListOfAuths, NSListOfAuths) char *CommaListOfAuths; char *NSListOfAuths;

Description

The MatchAllAuthsList subroutine compares the CommaListOfAuths against the NSListOfAuths. It returns a non-zero value if all the authorizations in CommaListOfAuths are contained in NSListOfAuths. The MatchAllAuths subroutine calls the MatchAllAuthsList subroutine passing in the results of the GetUserAuths subroutine in place of NSListOfAuths. If NSListOfAuths contains the OFF keyword, MatchAllAuthsList will return a zero value. If NSListOfAuths contains the ALL keyword and not the OFF keyword, MatchAllAuthsList will return a non-zero value.

The MatchAnyAuthsList subroutine compares the CommaListOfAuths against the NSListOfAuths. It returns a non-zero value if one or more of the authorizations in CommaListOfAuths are contained in NSListOfAuths. The MatchAnyAuths subroutine calls the MatchAnyAuthsList subroutine passing in the results of the GetUserAuths subroutine in place of NSListOfAuths. If NSListOfAuths contains the OFF keyword, MatchAnyAuthsList will return a zero value. If NSListOfAuths contains the ALL keyword and not the OFF keyword, MatchAnyAuthsList will return a non-zero value.

Parameters

Commal istOfAuths Specifies one or more authorizations, each separated by a comma. NSListOfAuths 1 4 1

Specifies zero or more authorizations. Each authorization is null terminated. The last

entry in the list must be a null string.

Return Values

The subroutines return a non-zero value if a proper match was found. Otherwise, they will return zero. If an error occurs, the subroutines will return zero and set errno to indicate the error. If the subroutine returns zero and no error occurred, errno is set to zero.

mblen Subroutine

Purpose

Determines the length in bytes of a multibyte character.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
int mblen( MbString, Number)
const char *MbString;
size t Number;
```

Description

The **mblen** subroutine determines the length, in bytes, of a multibyte character.

Parameters

Mbstring Points to a multibyte character string.

Specifies the maximum number of bytes to consider. Number

Return Values

The mblen subroutine returns 0 if the MbString parameter points to a null character. It returns -1 if a character cannot be formed from the number of bytes specified by the *Number* parameter. If *MbString* is a null pointer, 0 is returned.

Related Information

The "mbslen Subroutine" on page 674, "mbstowcs Subroutine" on page 680, and "mbtowc Subroutine" on page 681.

Subroutines, Example Programs, and Libraries, in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview and Multibyte Code and Wide Character Code Conversion Subroutines in AIX 5L Version 5.2 National Language Support Guide and Reference.

mbrlen Subroutine

Purpose

Get number of bytes in a character (restartable).

Library

Standard Library (libc.a)

Syntax

```
#include <wchar.h>
size_t mbrlen (const char *s, size_t n, mbstate_t *ps )
```

Description

If s is not a null pointer, **mbrlen** determines the number of bytes constituting the character pointed to by s. It is equivalent to:

```
mbstate t internal;
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);
```

If ps is a null pointer, the mbrlen function uses its own internal mbstate_t object, which is initialized at program startup to the initial conversion state. Otherwise, the mbstate t object pointed to by ps is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls mbrlen.

The behavior of this function is affected by the LC CTYPE category of the current locale.

Return Values

The **mbrlen** function returns the first of the following that applies:

If the next n or fewer bytes complete the character that corresponds to the null wide-character positive

If the next n or fewer bytes complete a valid character; the value returned is the number of

bytes that complete the character.

If the next n bytes contribute to an incomplete but potentially valid character, and all n bytes (size_t)-2

have been processed. When n has at least the value of the MB_CUR_MAX macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with

state-dependent encodings).

(size_t)-1 If an encoding error occurs, in which case the next n or fewer bytes do not contribute to a

complete and valid character. In this case, EILSEQ is stored in errno and the conversion state

is undefined.

Error Codes

The mbrlen function may fail if:

EINVAL **ps** points to an object that contains an invalid conversion state.

EILSEQ Invalid character sequence is detected.

Related Information

The **mbsinit** ("mbsinit Subroutine" on page 673) subroutine, **mbrtowc** ("mbrtowc Subroutine") subroutine.

mbrtowc Subroutine

Purpose

Convert a character to a wide-character code (restartable).

Library

Standard Library (libc.a)

Syntax

```
#include <wchar.h>
size_t mbrtowc (wchar_t * pwc, const char * s, size_t n, mbstate_t * ps);
```

Description

If s is a null pointer, the **mbrtowc** function is equivalent to the call:

```
mbrtowc(NULL, '''', 1, ps)
```

In this case, the values of the arguments **pwc** and **n** are ignored.

If s is not a null pointer, the **mbrtowc** function inspects at most n bytes beginning at the byte pointed to by s to determine the number of bytes needed to complete the next character (including any shift sequences). If the function determines that the next character is completed, it determines the value of the

corresponding wide-character and then, if pwc is not a null pointer, stores that value in the object pointed to by pwc. If the corresponding wide-character is the null wide-character, the resulting state described is the initial conversion state.

If ps is a null pointer, the mbrtowc function uses its own internal mbstate_t object, which is initialized at program startup to the initial conversion state. Otherwise, the **mbstate t** object pointed to by ps is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls mbrtowc.

The behavior of this function is affected by the LC CTYPE category of the current locale.

Return Values

The **mbrtowc** function returns the first of the following that applies:

0 If the next n or fewer bytes complete the character that corresponds to the null wide-character

(which is the value stored).

positive If the next n or fewer bytes complete a valid character (which is the value stored); the value

returned is the number of bytes that complete the character.

(size_t)-2 If the next n bytes contribute to an incomplete but potentially valid character, and all n bytes

> have been processed (no value is stored). When n has at least the value of the MB_CUR_MAX macro, this case can only occur if s points at a sequence of redundant shift sequences (for

implementations with state-dependent encodings).

If an encoding error occurs, in which case the next n or fewer bytes do not contribute to a (size_t)-1

complete and valid character (no value is stored). In this case, EILSEQ is stored in errno and

the conversion state is undefined.

Error Codes

The mbrtowc function may fail if:

EINVAL ps points to an object that contains an invalid conversion state.

Invalid character sequence is detected. **EILSEQ**

Related Information

The **mbsinit** ("mbsinit Subroutine" on page 673) subroutine.

mbsadvance Subroutine

Purpose

Advances to the next multibyte character.

Note: The mbsadvance subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

Standard C Library (libc.a)

Syntax

```
#include <mbstr.h>
```

```
char *mbsadvance (S)
const char *S;
```

The mbsadvance subroutine locates the next character in a multibyte character string. The LC_CTYPE category affects the behavior of the mbsadvance subroutine.

Parameters

Contains a multibyte character string.

Return Values

If the S parameter is not a null pointer, the **mbsadvance** subroutine returns a pointer to the next multibyte character in the string pointed to by the S parameter. The character at the head of the string pointed to by the S parameter is skipped. If the S parameter is a null pointer or points to a null string, a null pointer is returned.

Examples

To find the next character in a multibyte string, use the following:

```
#include <mbstr.h>
#include <locale.h>
#include <stdlib.h>
main()
   char *mbs, *pmbs;
   (void) setlocale(LC ALL, "");
   ** Let mbs point to the beginning of a multi-byte string.
   */
  pmbs = mbs;
  while(pmbs){
     pmbs = mbsadvance(mbs);
     /* pmbs points to the next multi-byte character
     ** in mbs */
```

Related Information

The **mbsinvalid** ("mbsinvalid Subroutine" on page 674) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

mbscat, mbscmp, or mbscpy Subroutine

Purpose

Performs operations on multibyte character strings.

Library

Standard C Library (libc.a)

Syntax

#include <mbstr.h>

```
char *mbscat(MbString1, MbString2)
char *MbString1, *MbString2;
int mbscmp(MbString1, MbString2)
char *MbString1, *MbString2;
char *mbscpy(MbString1, MbString2)
char *MbString1, *MbString2;
```

The **mbscat**, **mbscmp**, and **mbscpy** subroutines operate on null-terminated multibyte character strings.

The mbscat subroutine appends multibyte characters from the MbString2 parameter to the end of the MbString1 parameter, appends a null character to the result, and returns MbString1.

The mbscmp subroutine compares multibyte characters based on their collation weights as specified in the LC COLLATE category. The mbscmp subroutine compares the MbString1 parameter to the MbString2 parameter, and returns an integer greater than 0 if MbString1 is greater than MbString2. It returns 0 if the strings are equivalent and returns an integer less than 0 if MbString1 is less than MbString2.

The **mbscpy** subroutine copies multibyte characters from the *MbString2* parameter to the *MbString1* parameter and returns *MbString1*. The copy operation terminates with the copying of a null character.

Related Information

The **mbsncat**, **mbsncmp**, **mbsncpy** ("mbsncat, mbsncmp, or mbsncpy Subroutine" on page 675) subroutine, wcscat, wcscmp, wcscpy subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

mbschr Subroutine

Purpose

Locates a character in a multibyte character string.

Library

Standard C Library (libc.a)

Syntax

```
#include <mbstr.h>
char *mbschr( MbString, MbCharacter)
char *MbString;
mbchar_t MbCharacter;
```

Description

The **mbschr** subroutine locates the first occurrence of the value specified by the *MbCharacter* parameter in the string pointed to by the MbString parameter. The MbCharacter parameter specifies a multibyte character represented as an integer. The terminating null character is considered to be part of the string.

The **LC CTYPE** category affects the behavior of the **mbschr** subroutine.

Parameters

MbString Points to a multibyte character string.

MbCharacter Specifies a multibyte character represented as an integer.

Return Values

The **mbschr** subroutine returns a pointer to the value specified by the *MbCharacter* parameter within the multibyte character string, or a null pointer if that value does not occur in the string.

Related Information

The "mbspbrk Subroutine" on page 676, "mbsrchr Subroutine" on page 677, "mbstomb Subroutine" on page 679, wcschr subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

mbsinit Subroutine

Purpose

Determine conversion object status.

Library

Standard Library (libc.a)

Syntax

```
#include <wchar.h>
int mbsinit (const mbstate t * p);
```

Description

If ps is not a null pointer, the **mbsinit** function determines whether the object pointed to by ps describes an initial conversion state.

The **mbstate_t** object is used to describe the current conversion state from a particular character sequence to a wide-character sequence (or vice versa) under the rules of a particular setting of the LC_CTYPE category of the current locale.

The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new character sequence in the initial shift state. A zero valued mbstate t object is at least one way to describe an initial conversion state. A zero valued **mbstate** t object can be used to initiate conversion involving any character sequence, in any LC CTYPE category setting.

Return Values

The mbsinit function returns non-zero if ps is a null pointer, or if the pointed-to object describes an initial conversion state; otherwise, it returns zero.

If an **mbstate** t object is altered by any of the functions described as restartable, and is then used with a different character sequence, or in the other conversion direction, or with a different LC_CTYPE category setting than on earlier function calls, the behavior is undefined.

Related Information

The "mbrlen Subroutine" on page 668, "mbrtowc Subroutine" on page 669, **wctomb** subroutine, "mbsrtowcs Subroutine" on page 678, **wcsrtombs** subroutine.

mbsinvalid Subroutine

Purpose

Validates characters of multibyte character strings.

Note: The **mbsinvalid** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

Standard C Library (libc.a)

Syntax

```
#include <mbstr.h>
char *mbsinvalid ( S)
const char *S;
```

Description

The **mbsinvalid** subroutine examines the string pointed to by the S parameter to determine the validity of characters. The **LC_CTYPE** category affects the behavior of the **mbsinvalid** subroutine.

Parameters

S Contains a multibyte character string.

Return Values

The **mbsinvalid** subroutine returns a pointer to the byte following the last valid multibyte character in the S parameter. If all characters in the S parameter are valid multibyte characters, a null pointer is returned. If the S parameter is a null pointer, the behavior of the **mbsinvalid** subroutine is unspecified.

Related Information

The "mbsadvance Subroutine" on page 670.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference

mbslen Subroutine

Purpose

Determines the number of characters (code points) in a multibyte character string.

Note: The **mbslen** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

674 Technical Reference, Volume 1: Base Operating System and Extensions

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
size t mbslen( MbString)
char *mbs;
```

Description

The **mbslen** subroutine determines the number of characters (code points) in a multibyte character string. The **LC_CTYPE** category affects the behavior of the **mbslen** subroutine.

Parameters

MbString

Points to a multibyte character string.

Return Values

The **mbslen** subroutine returns the number of multibyte characters in a multibyte character string. It returns 0 if the MbString parameter points to a null character or if a character cannot be formed from the string pointed to by this parameter.

Related Information

The mblen ("mblen Subroutine" on page 667) subroutine, mbstowcs ("mbstowcs Subroutine" on page 680) subroutine, mbtowc ("mbtowc Subroutine" on page 681) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview and Multibyte Code and Wide Character Code Conversion Subroutines in AIX 5L Version 5.2 National Language Support Guide and Reference.

mbsncat, mbsncmp, or mbsncpy Subroutine

Purpose

Performs operations on a specified number of null-terminated multibyte characters.

Note: These subroutines are specific to the manufacturer. They are not defined in the POSIX, ANSI, or X/Open standards. Use of these subroutines may affect portability.

Library

Standard C Library (libc.a)

Syntax

```
#include <mbstr.h>
char *mbsncat(MbString1, MbString2, Number)
char * MbString1, * MbString2;
size_t Number;
```

```
int mbsncmp(MbString1, MbString2, Number)
char *MbString1, *MbString2;
size_t Number;
char *mbsncpy(MbString1, MbString2, Number)
char *MbString1, *MbString2;
size t Number;
```

The **mbsncat**, **mbsncmp**, and **mbsncpy** subroutines operate on null-terminated multibyte character strings.

The **mbsncat** subroutine appends up to the specified maximum number of multibyte characters from the *MbString2* parameter to the end of the *MbString1* parameter, appends a null character to the result, and then returns the *MbString1* parameter.

The **mbsncmp** subroutine compares the collation weights of multibyte characters. The **LC_COLLATE** category specifies the collation weights for all characters in a locale. The **mbsncmp** subroutine compares up to the specified maximum number of multibyte characters from the *MbString1* parameter to the *MbString2* parameter. It then returns an integer greater than 0 if *MbString1* is greater than *MbString2*. It returns 0 if the strings are equivalent. It returns an integer less than 0 if *MbString1* is less than *MbString2*.

The **mbsncpy** subroutine copies up to the value of the *Number* parameter of multibyte characters from the *MbString2* parameter to the *MbString1* parameter and returns *MbString1*. If *MbString2* is shorter than *Number* multi-byte characters, *MbString1* is padded out to *Number* characters with null characters.

Parameters

MbString1Contains a multibyte character string.MbString2Contains a multibyte character string.NumberSpecifies a maximum number of characters.

Related Information

The "mbscat, mbscmp, or mbscpy Subroutine" on page 671, "mbscat, mbscmp, or mbscpy Subroutine" on page 671, "mbscat, mbscmp, or mbscpy Subroutine" on page 671, **wcsncat** subroutine, **wcsncmp** subroutine, **wcsncpy** subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

mbspbrk Subroutine

Purpose

Locates the first occurrence of multibyte characters or code points in a string.

Note: The **mbspbrk** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

Standard C Library (libc.a)

Syntax

```
#include <mbstr.h>
char *mbspbrk( MbString1,  MbString2)
char *MbString1, *MbString2;
```

Description

The mbspbrk subroutine locates the first occurrence in the string pointed to by the MbString1 parameter, of any character of the string pointed to by the *MbString2* parameter.

Parameters

MbString1 Points to the string being searched. MbString2 Pointer to a set of characters in a string.

Return Values

The mbspbrk subroutine returns a pointer to the character. Otherwise, it returns a null character if no character from the string pointed to by the MbString2 parameter occurs in the string pointed to by the MbString1 parameter.

Related Information

The "mbschr Subroutine" on page 672, "mbsrchr Subroutine," "mbstomb Subroutine" on page 679, wcspbrk subroutine, wcswcs subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

mbsrchr Subroutine

Purpose

Locates a character or code point in a multibyte character string.

Library

Standard C Library (libc.a)

Syntax

```
#include <mbstr.h>
char *mbsrchr( MbString, MbCharacter)
char *MbString;
int MbCharacter;
```

Description

The **mbschr** subroutine locates the last occurrence of the *MbCharacter* parameter in the string pointed to by the MbString parameter. The MbCharacter parameter is a multibyte character represented as an integer. The terminating null character is considered to be part of the string.

Parameters

MbString Points to a multibyte character string.

MbCharacter Specifies a multibyte character represented as an integer.

Return Values

The **mbsrchr** subroutine returns a pointer to the *MbCharacter* parameter within the multibyte character string. It returns a null pointer if MbCharacter does not occur in the string.

Related Information

The "mbschr Subroutine" on page 672, "mbspbrk Subroutine" on page 676, "mbstomb Subroutine" on page 679, wcsrchr subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference

mbsrtowcs Subroutine

Purpose

Convert a character string to a wide-character string (restartable).

Library

Standard Library (libc.a)

Syntax

```
#include <wchar.h>
size_t mbsrtowcs ((wchar t * dst, const char ** src, size t len, mbstate t * ps) ;
```

Description

The **mbsrtowcs** function converts a sequence of characters, beginning in the conversion state described by the object pointed to by ps, from the array indirectly pointed to by src into a sequence of corresponding wide-characters. If dst is not a null pointer, the converted characters are stored into the array pointed to by dst. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops early in either of the following cases:

- When a sequence of bytes is encountered that does not form a valid character.
- When len codes have been stored into the array pointed to by dst (and dst is not a null pointer).

Each conversion takes place as if by a call to the **mbrtowc** function.

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last character converted (if any). If conversion stopped due to reaching a terminating null character, and if dst is not a null pointer, the resulting state described is the initial conversion state.

If ps is a null pointer, the mbsrtowcs function uses its own internal mbstate t object, which is initialised at program startup to the initial conversion state. Otherwise, the **mbstate t** object pointed to by ps is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls mbsrtowcs.

The behavior of this function is affected by the LC CTYPE category of the current locale.

Return Values

If the input conversion encounters a sequence of bytes that do not form a valid character, an encoding error occurs. In this case, the mbsrtowcs function stores the value of the macro EILSEQ in errno and returns (size t)-1); the conversion state is undefined. Otherwise, it returns the number of characters successfully converted, not including the terminating null (if any).

Error Codes

The **mbsrtowcs** function may fail if:

EINVAL ps points to an object that contains an invalid conversion state.

EILSEQ Invalid character sequence is detected.

Related Information

The "mbsinit Subroutine" on page 673, "mbrtowc Subroutine" on page 669.

mbstomb Subroutine

Purpose

Extracts a multibyte character from a multibyte character string.

Note: The mbstomb subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

Standard C Library (libc.a)

Syntax

#include <mbstr.h>

mbchar t mbstomb (MbString) const char *MbString;

Description

The **mbstomb** subroutine extracts the multibyte character pointed to by the *MbString* parameter from the multibyte character string. The **LC CTYPE** category affects the behavior of the **mbstomb** subroutine.

Parameters

MbString Contains a multibyte character string.

Return Values

The **mbstomb** subroutine returns the code point of the multibyte character as a **mbchar_t** data type. If an unusable multibyte character is encountered, a value of 0 is returned.

Related Information

The "mbschr Subroutine" on page 672, "mbspbrk Subroutine" on page 676, "mbsrchr Subroutine" on page 677.

Subroutines, Example Programs, and Libraries in *AIX 5L Version 5.2 General Programming Concepts:* Writing and Debugging Programs.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

mbstowcs Subroutine

Purpose

Converts a multibyte character string to a wide character string.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
size_t mbstowcs( WcString, String, Number)
wchar_t *WcString;
const char *String;
size t Number;
```

Description

The **mbstowcs** subroutine converts the sequence of multibyte characters pointed to by the *String* parameter to wide characters and places the results in the buffer pointed to by the *WcString* parameter. The multibyte characters are converted until a null character is reached or until the number of wide characters specified by the *Number* parameter have been processed.

Parameters

WcString Points to the area where the result of the conversion is stored.

String Points to a multibyte character string.

Number Specifies the maximum number of wide characters to be converted.

Return Values

The **mbstowcs** subroutine returns the number of wide characters converted, not including a null terminator, if any. If an invalid multibyte character is encountered, a value of -1 is returned. The *WcString* parameter does not include a null terminator if the value *Number* is returned.

If *WcString* is a null wide character pointer, the **mbstowcs** subroutine returns the number of elements required to store the wide character codes in an array.

Error Codes

The **mbstowcs** subroutine fails if the following occurs:

EILSEQ Invalid byte sequence is detected.

Related Information

The "mblen Subroutine" on page 667, "mbslen Subroutine" on page 674, "mbtowc Subroutine" on page 681, **wcstombs** subroutine, **wctomb** subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview for Programming and Multibyte Code and Wide Character Code Conversion Subroutines in AIX 5L Version 5.2 National Language Support Guide and Reference.

mbswidth Subroutine

Purpose

Determines the number of multibyte character string display columns.

Note: The mbswidth subroutine is specific to this manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

Standard C Library (libc.a)

Syntax

#include <mbstr.h>

int mbswidth (MbString, Number) const char *MbString; size t Number;

Description

The mbswidth subroutine determines the number of display columns required for a multibyte character string.

Parameters

MbString Contains a multibyte character string.

Number Specifies the number of bytes to read from the *s* parameter.

Return Values

The mbswidth subroutine returns the number of display columns that will be occupied by the MbString parameter if the number of bytes (specified by the *Number* parameter) read from the *MbString* parameter form valid multibyte characters. If the MbString parameter points to a null character, a value of 0 is returned. If the MbString parameter does not point to valid multibyte characters, -1 is returned. If the MbString parameter is a null pointer, the behavior of the mbswidth subroutine is unspecified.

Related Information

The wcswidth subroutine, wcwidth subroutine.

National Language Support Overview in AIX 5L Version 5.2 National Language Support Guide and Reference.

mbtowc Subroutine

Purpose

Converts a multibyte character to a wide character.

Library

Standard C Library (libc.a)

Syntax

```
#include <stdlib.h>
int mbtowc ( WideCharacter, String, Number)
wchar t *WideCharacter;
const char *String;
size t Number;
```

Description

The mbtowc subroutine converts a multibyte character to a wide character and returns the number of bytes of the multibyte character.

The **mbtowc** subroutine determines the number of bytes that comprise the multibyte character pointed to by the String parameter. It then converts the multibyte character to a corresponding wide character and, if the WideCharacter parameter is not a null pointer, places it in the location pointed to by the WideCharacter parameter. If the WideCharacter parameter is a null pointer, the mbtowc subroutine returns the number of converted bytes but does not change the WideCharacter parameter value. If the WideCharacter parameter returns a null value, the multibyte character is not converted.

Parameters

WideCharacter Specifies the location where a wide character is to be placed.

Strina Specifies a multibyte character.

Number Specifies the maximum number of bytes of a multibyte character.

Return Values

The **mbtowc** subroutine returns a value of 0 if the *String* parameter is a null pointer. The subroutine returns a value of -1 if the bytes pointed to by the String parameter do not form a valid multibyte character before the number of bytes specified by the *Number* parameter (or fewer) have been processed. It then sets the errno global variable to indicate the error. Otherwise, the number of bytes comprising the multibyte character is returned.

Error Codes

The **mbtowc** subroutine fails if the following occurs:

EILSEQ Invalid byte sequence is detected.

Related Information

The "mblen Subroutine" on page 667, "mbslen Subroutine" on page 674, "mbstowcs Subroutine" on page 680, wcstombs subroutine, wctomb subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview and Multibyte Code and Wide Character Code Conversion Subroutines in AIX 5L Version 5.2 National Language Support Guide and Reference.

memccpy, memchr, memcmp, memcpy, memset or memmove Subroutine

Purpose

Performs memory operations.

Library

Standard C Library (libc.a)

Syntax

```
#include <memory.h>
void *memccpy (Target, Source, C, N)
void *Target;
const void *Source;
int C;
size_t N;
void *memchr (S, C, N)
const void *S;
int C;
size t N;
int memcmp (Target, Source, N)
const void *Target, *Source;
size_t N;
void *memcpy (Target, Source, N)
void *Target;
const void *Source;
size t N;
void *memset (S, C, N)
void *S;
int C;
size_t N;
void *memmove (Target, Source, N)
void *Source;
const void *Target;
size_t //;
```

Description

The **memory** subroutines operate on memory areas. A memory area is an array of characters bounded by a count. The **memory** subroutines do not check for the overflow of any receiving memory area. All of the **memory** subroutines are declared in the **memory.h** file.

The memccpy subroutine copies characters from the memory area specified by the Source parameter into the memory area specified by the *Target* parameter. The **memccpy** subroutine stops after the first character specified by the C parameter (converted to the unsigned char data type) is copied, or after N characters are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.

The **memcmp** subroutine compares the first *N* characters as the **unsigned cha**r data type in the memory area specified by the Target parameter to the first N characters as the unsigned char data type in the memory area specified by the *Source* parameter.

The **memcpy** subroutine copies N characters from the memory area specified by the Source parameter to the area specified by the *Target* parameter and then returns the value of the *Target* parameter.

The **memset** subroutine sets the first N characters in the memory area specified by the S parameter to the value of character C and then returns the value of the S parameter.

Like the **memcpy** subroutine, the **memmove** subroutine copies *N* characters from the memory area specified by the *Source* parameter to the area specified by the *Target* parameter. However, if the areas of the *Source* and *Target* parameters overlap, the move is performed nondestructively, proceeding from right to left.

The **memccpy** subroutine is not in the ANSI C library.

Parameters

Target Points to the start of a memory area.

Source Points to the start of a memory area.

C Specifies a character to search.

N Specifies the number of characters to search.

S Points to the start of a memory area.

Return Values

The **memccpy** subroutine returns a pointer to character *C* after it is copied into the area specified by the *Target* parameter, or a null pointer if the *C* character is not found in the first *N* characters of the area specified by the *Source* parameter.

The **memchr** subroutine returns a pointer to the first occurrence of the C character in the first N characters of the memory area specified by the S parameter, or a null pointer if the C character is not found.

The **memcmp** subroutine returns the following values:

Less than 0 If the value of the *Target* parameter is less than the values of the *Source* parameter.
 Equal to 0 If the value of the *Target* parameter equals the value of the *Source* parameter.
 Greater than 0 If the value of the *Target* parameter is greater than the value of the *Source* parameter.

Related Information

The swab subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

mincore Subroutine

Purpose

Determines residency of memory pages.

Library

Standard C Library (libc.a).

Syntax

```
int mincore ( addr, len, * vec)
caddr t addr;
size t len;
char *vec;
```

Description

The mincore subroutine returns the primary-memory residency status for regions created from calls made to the **mmap** ("mmap or mmap64 Subroutine" on page 690) subroutine. The status is returned as a character for each memory page in the range specified by the addr and len parameters. The least significant bit of each character returned is set to 1 if the referenced page is in primary memory. Otherwise, the bit is set to 0. The settings of the other bits in each character are undefined.

Parameters

addr Specifies the starting address of the memory pages whose residency is to be determined. Must be a multiple of the page size returned by the sysconf subroutine using the SC PAGE SIZE value for the Name parameter.

len Specifies the length, in bytes, of the memory region whose residency is to be determined. If the len value is not a multiple of the page size as returned by the sysconf subroutine using the SC PAGE SIZE value for the Name parameter, the length of the region is rounded up to the next multiple of the page size.

Specifies the character array where the residency status is returned. The system assumes that the character vec array specified by the vec parameter is large enough to encompass a returned character for each page specified.

Return Values

When successful, the mincore subroutine returns 0. Otherwise, it returns -1 and sets the errno global variable to indicate the error.

Error Codes

If the mincore subroutine is unsuccessful, the errno global variable is set to one of the following values:

EFAULT A part of the buffer pointed to by the vec parameter is out of range or otherwise inaccessible.

EINVAL The addr parameter is not a multiple of the page size as returned by the sysconf subroutine using the

_SC_PAGE_SIZE value for the Name parameter.

ENOMEM Addresses in the (addr, addr + len) range are invalid for the address space of the process, or specify one

or more pages that are not mapped.

Related Information

The **mmap** ("mmap or mmap64 Subroutine" on page 690) subroutine, **sysconf** subroutine.

List of Memory Manipulation Services in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

mkdir Subroutine

Purpose

Creates a directory.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/stat.h>
int mkdir ( Path, Mode)
const char *Path;
mode_t Mode;
```

Description

The **mkdir** subroutine creates a new directory.

The new directory has the following:

- The owner ID is set to the process-effective user ID.
- If the parent directory has the SetFileGroupID (S_ISGID) attribute set, the new directory inherits the group ID of the parent directory. Otherwise, the group ID of the new directory is set to the effective group ID of the calling process.
- Permission and attribute bits are set according to the value of the Mode parameter, with the following modifications:
 - All bits set in the process-file mode-creation mask are cleared.
 - The SetFileUserID and Sticky (S_ISVTX) attributes are cleared.
- If the *Path* variable names a symbolic link, the link is followed. The new directory is created where the variable pointed.

Parameters

Path Specifies the name of the new directory. If Network File System (NFS) is installed on your

system, this path can cross into another node. In this case, the new directory is created at that

node.

To execute the **mkdir** subroutine, a process must have search permission to get to the parent

directory of the Path parameter as well as write permission in the parent directory itself.

Mode Specifies the mask for the read, write, and execute flags for owner, group, and others. The

Mode parameter specifies directory permissions and attributes. This parameter is constructed

by logically ORing values described in the sys/mode.h file.

Return Values

Upon successful completion, the **mkdir** subroutine returns a value of 0. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **mkdir** subroutine is unsuccessful and the directory is not created if one or more of the following are true:

EACCES Creating the requested directory requires writing in a directory with a

mode that denies write permission.

EEXIST The named file already exists.

EROFS The named file resides on a read-only file system.

ENOSPC The file system does not contain enough space to hold the contents of

the new directory or to extend the parent directory of the new directory.

EMLINK The link count of the parent directory exceeds the maximum

(LINK_MAX) number. (LINK_MAX) is defined in limits.h file.

ENAMETOOLONG The *Path* parameter or a path component is too long and cannot be

truncated.

ENOENT A component of the path prefix does not exist or the Path parameter

points to an empty string.

ENOTDIR A component of the path prefix is not a directory.

EDQUOT The directory in which the entry for the new directory is being placed cannot be extended, or an i-node or disk blocks could not be allocated

for the new directory because the user's or group's quota of disk blocks or i-nodes on the file system containing the directory is exhausted.

The **mkdir** subroutine can be unsuccessful for other reasons. See "Appendix A. Base Operating System" Error Codes for Services That Require Path-Name Resolution" for a list of additional errors.

If NFS is installed on the system, the mkdir subroutine is also unsuccessful if the following is true:

ETIMEDOUT The connection timed out.

Related Information

The **chmod** ("chmod or fchmod Subroutine" on page 124) subroutine, **mknod** ("mknod or mkfifo Subroutine") subroutine, rmdir subroutine, umask subroutine.

The **chmod** command, **mkdir** command, **mknod** command.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

mknod or mkfifo Subroutine

Purpose

Creates an ordinary file, first-in-first-out (FIFO), or special file.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/stat.h>
int mknod (const char * Path, mode t Mode, dev t Device)
char *Path;
int Mode;
dev t Device;
int mkfifo (const char *Path, mode t Mode)
const char *Path;
int Mode;
```

Description

The **mknod** subroutine creates a new regular file, special file, or FIFO file. Using the **mknod** subroutine to create file types (other than FIFO or special files) requires root user authority.

For the **mknod** subroutine to complete successfully, a process must have both search and write permission in the parent directory of the Path parameter.

The mkfifo subroutine is an interface to the mknod subroutine, where the new file to be created is a FIFO or special file. No special system privileges are required.

The new file has the following characteristics:

- File type is specified by the Mode parameter.
- Owner ID is set to the effective user ID of the process.
- Group ID of the file is set to the group ID of the parent directory if the SetGroupID attribute (S_ISGID) of the parent directory is set. Otherwise, the group ID of the file is set to the effective group ID of the calling process.
- · Permission and attribute bits are set according to the value of the Mode parameter. All bits set in the file-mode creation mask of the process are cleared.

Upon successful completion, the mkfifo subroutine marks for update the st atime, st ctime, and st mtime fields of the file. It also marks for update the st ctime and st mtime fields of the directory that contains the new entry.

If the new file is a character special file having the **S_IMPX** attribute (multiplexed character special file), when the file is used, additional path-name components can appear after the path name as if it were a directory. The additional part of the path name is available to the device driver of the file for interpretation. This feature provides a multiplexed interface to the device driver.

Parameters

Path Names the new file. If Network File System (NFS) is installed on your system, this path can cross into another node.

Mode Specifies the file type, attributes, and access permissions. This parameter is constructed by logically

ORing values described in the sys/mode.h file. Device

Specifies the ID of the device, which corresponds to the st rdev member of the structure returned by the statx subroutine. This parameter is configuration-dependent and used only if the Mode parameter specifies a block or character special file. If the file you specify is a remote file, the value of the Device parameter must be meaningful on the node where the file resides.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **mknod** subroutine fails and the new file is not created if one or more of the following are true:

EEXIST The named file exists.

EDQUOT The directory in which the entry for the new file is being placed cannot be extended, or an

i-node could not be allocated for the file because the user's or group's quota of disk blocks

or i-nodes on the file system is exhausted.

EISDIR The *Mode* parameter specifies a directory. Use the **mkdir** subroutine instead.

ENOSPC The directory that would contain the new file cannot be extended, or the file system is out of

file-allocation resources.

EPERM The Mode parameter specifies a file type other than S_IFIFO, and the calling process does

not have root user authority.

EROFS The directory in which the file is to be created is located on a read-only file system.

The mknod and mkfifo subroutine can be unsuccessful for other reasons. See "Appendix. A Base Operating System Error Codes for Services That Require Path-Name Resolution" (Appendix A, "Base Operating System Error Codes for Services That Require Path-Name Resolution," on page 1001) for a list of additional errors.

If NFS is installed on the system, the **mknod** subroutine can also fail if the following is true:

ETIMEDOUT The connection timed out.

Related Information

The chmod ("chmod or fchmod Subroutine" on page 124) subroutine, mkdir ("mkdir Subroutine" on page 685) subroutine, open, openx, or creat ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine, **statx** subroutine, **umask** subroutine.

The **chmod** command, **mkdir** command, **mknod** command.

The mode.h file, types.h file.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

mktemp or mkstemp Subroutine

Purpose

Constructs a unique file name.

Libraries

Standard C Library (libc.a)

Berkeley Compatibility Library (libbsd.a)

Syntax

```
#include <stdlib.h>
char *mktemp ( Template)
char *Template;
int mkstemp ( Template)
char *Template;
```

Description

The **mktemp** subroutine replaces the contents of the string pointed to by the *Template* parameter with a unique file name.

Note: The mktemp subroutine creates a filename and checks to see if the file exist. It that file does not exist, the name is returned. If the user calls mktemp twice without creating a file using the name returned by the first call to mktemp, then the second mktemp call may return the same name as the first **mktemp** call since the name does not exist.

To avoid this, either create the file after calling **mktemp** or use the **mkstemp** subroutine. The **mkstemp** subroutine creates the file for you.

To get the BSD version of this subroutine, compile with Berkeley Compatibility Library (libbsd.a).

The **mkstemp** subroutine performs the same substitution to the template name and also opens the file for reading and writing.

In BSD systems, the **mkstemp** subroutine was intended to avoid a race condition between generating a temporary name and creating the file. Because the name generation in the operating system is more random, this race condition is less likely. BSD returns a file name of / (slash).

Former implementations created a unique name by replacing X's with the process ID and a unique letter.

Parameters

Template

Points to a string to be replaced with a unique file name. The string in the *Template* parameter is a file name with up to six trailing X's. Since the system randomly generates a six-character string to replace the X's, it is recommended that six trailing X's be used.

Return Values

Upon successful completion, the **mktemp** subroutine returns the address of the string pointed to by the *Template* parameter.

If the string pointed to by the *Template* parameter contains no X's, and if it is an existing file name, the *Template* parameter is set to a null character, and a null pointer is returned; if the string does not match any existing file name, the exact string is returned.

Upon successful completion, the **mkstemp** subroutine returns an open file descriptor. If the **mkstemp** subroutine fails, it returns a value of -1.

Related Information

The **getpid** ("getpid, getpgrp, or getppid Subroutine" on page 345) subroutine, **tmpfile** subroutine, **tmpnam** or **tempnam** subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

mmap or mmap64 Subroutine

Purpose

Maps a file-system object into virtual memory.

Library

Standard C library (libc.a)

Syntax

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

void *mmap (addr, len, prot, flags, fildes, off)
void * addr;
size_t len;
int prot, flags, fildes;
off_t off;

void *mmap64 (addr, len, prot, flags, fildes, off)
void * addr:
```

```
size_t len;
int prot, flags, fildes;
off64_t off;
```

Attention: A file-system object should not be simultaneously mapped using both the **mmap** and **shmat** subroutines. Unexpected results may occur when references are made beyond the end of the object.

The mmap subroutine creates a new mapped file or anonymous memory region by establishing a mapping between a process-address space and a file-system object. Care needs to be taken when using the **mmap** subroutine if the program attempts to map itself. If the page containing executing instructions is currently referenced as data through an mmap mapping, the program will hang. Use the -H4096 binder option, and that will put the executable text on page boundaries. Then reset the file that contains the executable material, and view via an mmap mapping.

A region created by the mmap subroutine cannot be used as the buffer for read or write operations that involve a device. Similarly, an mmap region cannot be used as the buffer for operations that require either a pin or xmattach operation on the buffer.

Modifications to a file-system object are seen consistently, whether accessed from a mapped file region or from the read or write subroutine.

Child processes inherit all mapped regions from the parent process when the fork subroutine is called. The child process also inherits the same sharing and protection attributes for these mapped regions. A successful call to any exec subroutine will unmap all mapped regions created with the mmap subroutine.

The mmap64 subroutine is identical to the mmap subroutine except that the starting offset for the file mapping is specified as a 64-bit value. This permits file mappings which start beyond **OFF MAX**.

In the large file enabled programming environment, mmap is redefined to be mmap64.

If the application has requested SPEC1170 compliant behavior then the st_atime field of the mapped file is marked for update upon successful completion of the **mmap** call.

If the application has requested SPEC1170 compliant behavior then the st_ctime and st_mtime fields of a file that is mapped with MAP_SHARED and PROT_WRITE are marked for update at the next call to msync subroutine or munmap subroutine if the file has been modified.

Parameters

addr

Specifies the starting address of the memory region to be mapped. When the MAP FIXED flag is specified, this address must be a multiple of the page size returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter. A region is never placed at address zero, or at an address where it would overlap an existing region.

len

Specifies the length, in bytes, of the memory region to be mapped. The system performs mapping operations over whole pages only. If the len parameter is not a multiple of the page size, the system will include in any mapping operation the address range between the end of the region and the end of the page containing the end of the region.

prot

Specifies the access permissions for the mapped region. The sys/mman.h file defines the following access options:

PROT_READ

Region can be read.

PROT_WRITE

Region can be written.

PROT EXEC

Region can be executed.

PROT NONE

Region cannot be accessed.

The prot parameter can be the PROT NONE flag, or any combination of the PROT READ flag, PROT_WRITE flag, and PROT_EXEC flag logically ORed together. If the PROT_NONE flag is not specified, access permissions may be granted to the region in addition to those explicitly requested. However, write access will not be granted unless the PROT_WRITE flag is specified.

Note: The operating system generates a SIGSEGV signal if a program attempts an access that exceeds the access permission given to a memory region. For example, if the PROT_WRITE flag is not specified and a program attempts a write access, a SIGSEGV signal results.

If the region is a mapped file that was mapped with the MAP_SHARED flag, the mmap subroutine grants read or execute access permission only if the file descriptor used to map the file was opened for reading. It grants write access permission only if the file descriptor was opened for writing.

If the region is a mapped file that was mapped with the MAP_PRIVATE flag, the mmap subroutine grants read, write, or execute access permission only if the file descriptor used to map the file was opened for reading. If the region is an anonymous memory region, the mmap subroutine grants all requested access permissions.

fildes

Specifies the file descriptor of the file-system object to be mapped. If the MAP ANONYMOUS flag is set, the fildes parameter must be -1. After the successful completion of the mmap subroutine, the file specified by the fildes parameter may be closed without effecting the mapped region or the contents of the mapped file. Each mapped region creates a file reference, similar to an open file descriptor, which prevents the file data from being deallocated.

Note: The mmap subroutine supports the mapping of regular files only. An mmap call that specifies a file descriptor for a special file fails, returning the ENODEV error. An example of a file descriptor for a special file is one that might be used for mapping either I/O or device memory.

Specifies the file byte offset at which the mapping starts. This offset must be a multiple of the page size returned by the sysconf subroutine using the SC PAGE SIZE value for the Name parameter.

off

flags

Specifies attributes of the mapped region. Values for the *flags* parameter are constructed by a bitwise-inclusive ORing of values from the following list of symbolic names defined in the sys/mman.h

MAP FILE

Specifies the creation of a new mapped file region by mapping the file associated with the fildes file descriptor. The mapped region can extend beyond the end of the file, both at the time when the mmap subroutine is called and while the mapping persists. This situation could occur if a file with no contents was created just before the call to the mmap subroutine, or if a file was later truncated. However, references to whole pages following the end of the file result in the delivery of a SIGBUS signal. Only one of the MAP_FILE and MAP_ANONYMOUS flags must be specified with the mmap subroutine.

MAP ANONYMOUS

Specifies the creation of a new, anonymous memory region that is initialized to all zeros. This memory region can be shared only with the descendants of the current process. When using this flag, the fildes parameter must be -1. Only one of the MAP_FILE and MAP_ANONYMOUS flags must be specified with the mmap subroutine.

MAP_ VARIABLE

Specifies that the system select an address for the new memory region if the new memory region cannot be mapped at the address specified by the addr parameter, or if the addr parameter is null. Only one of the MAP_VARIABLE and MAP_FIXED flags must be specified with the mmap subroutine.

MAP_FIXED

Specifies that the mapped region be placed exactly at the address specified by the addr parameter. If the application has requested SPEC1170 complaint behavior and the mmap request is successful, the mapping replaces any previous mappings for the process' pages in the specified range. If the application has not requested SPEC1170 compliant behavior and a previous mapping exists in the range then the request fails. Only one of the MAP_VARIABLE and MAP_FIXED flags must be specified with the mmap subroutine.

MAP SHARED

When the MAP SHARED flag is set, modifications to the mapped memory region will be visible to other processes that have mapped the same region using this flag. If the region is a mapped file region, modifications to the region will be written to the file.

You can specify only one of the MAP_SHARED or MAP_PRIVATE flags with the mmap subroutine. MAP PRIVATE is the default setting when neither flag is specified unless you request SPEC1170 compliant behavior. In this case, you must choose either MAP SHARED or MAP_PRIVATE.

MAP PRIVATE

When the MAP_PRIVATE flag is specified, modifications to the mapped region by the calling process are not visible to other processes that have mapped the same region. If the region is a mapped file region, modifications to the region are not written to the file.

If this flag is specified, the initial write reference to an object page creates a private copy of that page and redirects the mapping to the copy. Until then, modifications to the page by processes that have mapped the same region with the MAP SHARED flag are visible.

You can specify only one of the MAP SHARED or MAP PRIVATE flags with the mmap subroutine. MAP_PRIVATE is the default setting when neither flag is specified unless you request SPEC1170 compliant behavior. In this case, you must choose either MAP_SHARED or MAP_PRIVATE.

Return Values

If successful, the mmap subroutine returns the address at which the mapping was placed. Otherwise, it returns -1 and sets the errno global variable to indicate the error.

Error Codes

Under the following conditions, the mmap subroutine fails and sets the errno global variable to:

EACCES The file referred to by the *fildes* parameter is not open for read access, or the file is not open for

write access and the PROT WRITE flag was specified for a MAP SHARED mapping operation.

Or, the file to be mapped has enforced locking enabled and the file is currently locked.

EBADF The *fildes* parameter is not a valid file descriptor, or the **MAP_ANONYMOUS** flag was set and the

fildes parameter is not -1.

EFBIG The mapping requested extends beyond the maximum file size associated with *fildes*.

EINVAL The *flags* or *prot* parameter is invalid, or the *addr* parameter or *off* parameter is not a multiple of

the page size returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name

parameter.

EINVAL The application has requested SPEC1170 compliant behavior and the value of flags is invalid

(neither MAP_PRIVATE nor MAP_SHARED is set).

EMFILE The application has requested SPEC1170 compliant behavior and the number of mapped regions

would excedd and implementation-dependent limit (per process or per system).

ENODEV The *fildes* parameter refers to an object that cannot be mapped, such as a terminal.

ENOMEM There is not enough address space to map *len* bytes, or the application has not requested Single

UNIX Specification, Version 2 compliant behavior and the MAP_FIXED flag was set and part of the

address-space range (addr, addr+len) is already allocated.

ENXIO The addresses specified by the range (off, off+len) are invalid for the fildes parameter.

EOVERFLOW The mapping requested extends beyond the offset maximum for the file description associated with

fildes

Related Information

The **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutine, **fork** ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, **munmap** ("munmap Subroutine" on page 726) subroutine, **read** subroutine, **shmat** subroutine, **sysconf** subroutine, **write** subroutine.

The pin kernel service, xmattach kernel service.

List of Memory Manipulation Services, List of Memory Mapping Services, Understanding Memory Mapping in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

mntctl Subroutine

Purpose

Returns information about the mount status of the system.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/types.h>
#include <sys/mntctl.h>
#include <sys/vmount.h>

int mntctl ( Command, Size, Buffer)
int Command;
int Size;
char *Buffer;
```

The **mntctl** subroutine is used to query the status of virtual file systems (also known as *mounted* file systems).

Each virtual file system (VFS) is described by a **vmount** structure. This structure is supplied when the VFS is created by the **vmount** subroutine. The **vmount** structure is defined in the **sys/vmount.h** file.

Parameters

Command Specifies the operation to be performed. Valid commands are defined in the sys/vmount.h file. At

present, the only command is:

MCTL QUERY

Query mount information.

Buffer Points to a data area that will contain an array of vmount structures. This data area holds the

information returned by the query command. Since the **vmount** structure is variable-length, it is necessary to reference the vmt length field of each structure to determine where in the *Buffer* area the

next structure begins.

Size Specifies the length, in bytes, of the buffer pointed to by the Buffer parameter.

Return Values

If the **mntctl** subroutine is successful, the number of **vmount** structures copied into the *Buffer* parameter is returned. If the *Size* parameter indicates the supplied buffer is too small to hold the **vmount** structures for all the current VFSs, the **mntctl** subroutine sets the first word of the *Buffer* parameter to the required size (in bytes) and returns the value 0. If the **mntctl** subroutine otherwise fails, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **mntctl** subroutine fails and the requested operation is not performed if one or both of the following are true:

EINVAL The *Command* parameter is not **MCTL_QUERY**, or the *Size* parameter is not a positive value.

EFAULT The *Buffer* parameter points to a location outside of the allocated address space of the process.

Related Information

The uvmount or umount subroutine, vmount or mount subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

modf, modff, or modfl Subroutine

Purpose

Decomposes a floating-point number.

Syntax

```
#include <math.h>
float modff (x, iptr)
float x;
float *iptr;
double modf (x, iptr)
```

```
double x, *iptr;
long double modfl (x, iptr)
long double x, *iptr;
```

The **modf**, **modf**, and **modfl** subroutines break the *x* parameter into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a floating-point value in the object pointed to by *iptr*.

Parameters

x Specifies the value to be computed.

iptr Points to the object where the integral part is stored.

Return Values

Upon successful completion, the **modf**, **modf**, and **mofl** subroutines return the signed fractional part of x.

If x is NaN, a NaN is returned, and *iptr is set to a NaN.

If x is $\pm \ln f$, ± 0 is returned, and *iptr is set to $\pm \ln f$.

Related Information

"class, _class, finite, isnan, or unordered Subroutines" on page 138 and "ldexp, ldexpf, or ldexpl Subroutine" on page 591

math.h in AIX 5L Version 5.2 Files Reference.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

128-Bit long Double Floating-Point Format in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

moncontrol Subroutine

Purpose

Starts and stops execution profiling after initialization by the monitor subroutine.

Library

Standard C Library (libc.a)

Syntax

```
#include <mon.h>
int moncontrol ( Mode)
int Mode:
```

Description

The **moncontrol** subroutine starts and stops profiling after profiling has been initialized by the **monitor** subroutine. It may be used with either **-p** or **-pg** profiling. When **moncontrol** stops profiling, no output data

file is produced. When profiling has been started by the monitor subroutine and the exit subroutine is called, or when the monitor subroutine is called with a value of 0, then profiling is stopped, and an output file is produced, regardless of the state of profiling as set by the moncontrol subroutine.

The moncontrol subroutine examines global and parameter data in the following order:

- 1. When the _mondata.prof_type global variable is neither -1 (-p profiling defined) nor +1 (-pg profiling defined), no action is performed, 0 is returned, and the function is considered complete.
 - The global variable is set to -1 in the mcrt0.o file and to +1 in the gcrt0.o file and defaults to 0 when the crt0.o file is used.
- 2. When the *Mode* parameter is 0, profiling is stopped. For any other value, profiling is started. The following global variables are used in a call to the **profil** ("profil Subroutine" on page 876) subroutine:

mondata.ProfBuf Buffer address

mondata.ProfBufSiz Buffer size/multirange flag mondata.ProfLoPC PC offset for hist buffer - I/O limit mondata.ProfScale PC scale/compute scale flag.

These variables are initialized by the **monitor** subroutine each time it is called to start profiling.

Parameters

Mode Specifies whether to start (resume) or stop profiling.

Return Values

The moncontrol subroutine returns the previous state of profiling. When the previous state was STOPPED, a 0 is returned. When the previous state was STARTED, a 1 is returned.

Error Codes

When the moncontrol subroutine detects an error from the call to the profil subroutine, a -1 is returned.

Related Information

The monitor ("monitor Subroutine") subroutine, monstartup ("monstartup Subroutine" on page 703) subroutine, profil ("profil Subroutine" on page 876) subroutine.

List of Memory Manipulation Services in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

monitor Subroutine

Purpose

Starts and stops execution profiling using data areas defined in the function parameters.

Library

Standard C Library (libc.a)

Syntax

```
#include <mon.h>
int monitor ( LowProgramCounter, HighProgramCounter, Buffer, BufferSize, NFunction)

OR
int monitor ( NotZeroA, DoNotCareA, Buffer,-1, NFunction)

OR
int monitor((caddr_t)0)

caddr_t LowProgramCounter, HighProgramCounter;
HISTCOUNTER *Buffer;
int BufferSize, NFunction;
caddr t NotZeroA, DoNotCareA;
```

Description

The **monitor** subroutine initializes the buffer area and starts profiling, or else stops profiling and writes out the accumulated profiling data. Profiling, when started, causes periodic sampling and recording of the program location within the program address ranges specified. Profiling also accumulates function call count data compiled with the **-p** or **-pg** option.

Executable programs created with the **cc -p** or **cc -pg** command automatically include calls to the **monitor** subroutine (through the **monstartup** and **exit** subroutines) to profile the complete user program, including system libraries. In this case, you do not need to call the **monitor** subroutine.

The **monitor** subroutine is called by the **monstartup** subroutine to begin profiling and by the **exit** subroutine to end profiling. The **monitor** subroutine requires a global data variable to define which kind of profiling, **-p** or **-pg**, is in effect. The **monitor** subroutine initializes four global variables that are used as parameters to the **profil** subroutine by the **moncontrol** subroutine:

- The monitor subroutine calls the moncontrol subroutine to start the profiling data gathering.
- The **moncontrol** subroutine calls the **profil** subroutine to start the system timer-driven program address sampling.
- The **prof** command processes the data file produced by **-p** profiling.
- The **gprof** command processes the data file produced by **-pg** profiling.

The monitor subroutine examines the global data and parameter data in this order:

- 1. When the _mondata.prof_type global variable is neither -1 (-p profiling defined) nor +1 (-pg profiling defined), an error is returned, and the function is considered complete.
 - The global variable is set to -1 in the **mcrt0.o** file and to +1 in the **gcrt0.o** file, and defaults to 0 when the **crt0.o** file is used.
- 2. When the first parameter to the **monitor** subroutine is 0, profiling is stopped and the data file is written out.
 - If **-p** profiling was in effect, then the file is named **mon.out**. If **-pg** profiling was in effect, the file is named **qmon.out**. The function is complete.
- 3. When the first parameter to the **monitor** subroutine is not , the **monitor** parameters and the profiling global variable, **_mondata.prof_type**, are examined to determine how to start profiling.
- 4. When the *BufferSize* parameter is not -1, a single program address range is defined for profiling, and the first **monitor** definition in the syntax is used to define the single program range.

5. When the *BufferSize* parameter is -1, multiple program address ranges are defined for profiling, and the second **monitor** definition in the syntax is used to define the multiple ranges. In this case, the *ProfileBuffer* value is the address of an array of **prof** structures. The size of the **prof** array is denoted by a zero value for the *HighProgramCounter* (p_high) field of the last element of the array. Each element in the array, except the last, defines a single programming address range to be profiled. Programming ranges must be in ascending order of the program addresses with ascending order of the **prof** array index. Program ranges may not overlap.

The buffer space defined by the p_buff and p_bufsize fields of all of the **prof** entries must define a single contiguous buffer area. Space for the function-count data is included in the first range buffer. Its size is defined by the *NFunction* parameter. The p_scale entry in the **prof** structure is ignored. The **prof** structure is defined in the**mon.h** file. It contains the following fields:

Parameters

LowProgramCounter
(prof name: p_low)

HighProgramCounter
(prof name: p high)

Buffer (prof name: p buff)

BufferSize
(prof name: p bufsize)

Defines the lowest execution-time program address in the range to be profiled. The value of the *LowProgramCounter* parameter cannot be 0 when using the**monitor** subroutine to begin profiling.

Defines the next address after the highest-execution time program address in the range to be profiled.

The program address parameters may be defined by function names or address expressions. If defined by a function name, then a function name expression must be used to dereference the function pointer to get the address of the first instruction in the function. This is required because the function reference in this context produces the address of the function descriptor. The first field of the descriptor is the address of the function code. See the examples for typical expressions to use. Defines the beginning address of an array of *BufferSize* HISTCOUNTERs to be used for data collection. This buffer includes the space for the program address-sampling counters and the function-count data areas. In the case of a multiple range specification, the space for the function-count data area is included at the beginning of the first range in the BufferSize specification. Defines the size of the buffer in number of HISTCOUNTERs. Each counter is of type HISTCOUNTER (defined as short in the **mon.h** file). When the buffer includes space for the function-count data area (single range specification and first range of a multi-range specification) the NFunction parameter defines the space to be used for the function count data, and the remainder is used for program-address sampling counters for the range defined. The scale for the profil call is calculated from the number of counters available for program address-sample counting and the address range defined by the LowProgramCounter and HighProgramCounter

parameters. See themon.h file.

NFunction

Defines the size of the space to be used for the function-count data area. The space is included as part of the first (or only) range buffer.

When **-p** profiling is defined, the *NFunction* parameter defines the maximum number of functions to be counted. The space required for each function is defined to be:

```
sizeof(struct poutcnt)
```

The **poutcnt** structure is defined in the **mon.h** file. The total function-count space required is:

```
NFunction * sizeof(struct poutcnt)
```

When **-pg** profiling is defined, the *NFunction* parameter defines the size of the space (in bytes) available for the function-count data structures, as follows:

```
range = HighProgramCounter - LowProgramCounter;
tonum = TO NUM ELEMENTS( range );
if ( tonum < MINARCS ) tonum = MINARCS;</pre>
if ( tonum > TO_MAX-1 ) tonum = TO_MAX-1;
tosize = tonum * sizeof( struct tostruct );
fromsize = FROM STG SIZE( range );
rangesize = tosize + fromsize + sizeof(struct
gfctl);
```

This is computed and summed for all defined ranges. In this expression, the functions and variables in capital letters as well as the structures are defined in the mon.h

Specifies a value of parameter 1, which is any value except 0. Ignored when it is not zero.

Specifies a value of parameter 2, of any value, which is ignored.

NotZeroA

DoNotCareA

Return Values

The **monitor** subroutine returns 0 upon successful completion.

Error Codes

If an error is found, the monitor subroutine sends an error message to stderr and returns -1.

Examples

1. This example shows how to profile the main load module of a program with -p profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
extern caddr_t etext; /*system end of main module text symbol*/
extern int start(); /*first function in main program*/
extern struct monglobal mondata; /*profiling global variables*/
struct desc {
    caddr_t begin;
    caddr_t toc;
    caddr_t env;
    /*environment pointer*/
}
} ;
                     /*function descriptor structure*/
struct desc *fd;
                       /*pointer to function descriptor*/
int rc;
                      /*monitor return code*/
```

```
int range:
                       /*program address range for profiling*/
int numfunc;
                      /*number of functions*/
HISTCOUNTER *buffer;
                       /*buffer address*/
                   /*number of program address sample counters*/
int numtics;
int BufferSize; /*total buffer size in numbers of HISTCOUNTERs*/
fd = (struct desc*)start; /*init descriptor pointer to start\
function*/
numfunc = 300;
                        /*arbitrary number for example*/
range = etext - fd->begin; /*compute program address range*/
numtics =NUM_HIST_COUNTERS(range); /*one counter for each 4 byte\
inst*/
BufferSize = numtics + ( numfunc*sizeof (struct poutcnt) \
HIST COUNTER SIZE );
                        /*allocate buffer space*/
buffer = (HISTCOUNTER *) malloc (BufferSize * HIST COUNTER SIZE);
if ( buffer == NULL ) /*didn't get space, do error recovery\
here*/
   return(-1);
mondata.prof type = PROF TYPE IS P; /*define -p profiling*/
rc = monitor( fd->begin, (caddr_t)etext, buffer, BufferSize, \
numfunc);
/*start*/
if ( rc != 0 ) /*profiling did not start, do error recovery\
here*/
   return(-1);
/*other code for analysis*/
rc = monitor( (caddr_t)0); /*stop profiling and write data file\
mon.out*/
if ( rc != 0 ) /*did not stop correctly, do error recovery here*/
   return (-1);
```

2. This example profiles the main program and the libc.a shared library with -p profiling. The range of addresses for the shared libc.a is assumed to be:

```
1ow = d0300000
high = d0312244
```

These two values can be determined from the loadquery subroutine at execution time, or by using a debugger to view the loaded programs' execution addresses and the loader map.

```
#include <sys/types.h>
#include <mon.h>
main()
extern caddr t etext; /*system end of text symbol*/
extern int start(); /*first function in main program*/
extern struct monglobal mondata; /*profiling global variables*/
struct prof pb[3]; /*prof array of 3 to define 2 ranges*/
int rc;
                     /*monitor return code*/
int range;
                    /*program address range for profiling*/
                   /*number of functions to count (max)*/
int numfunc;
int numtics;
                     /*number of sample counters*/
int num4fcnt; /*number of HISTCOUNTERs used for fun cnt space*/
int BufferSize1;
                    /*first range BufferSize*/
int BufferSize2;
                     /*second range BufferSize*/
caddr_t liblo=0xd0300000; /*lib low address (example only)*/
caddr_t libhi=0xd0312244; /*lib high address (example only)*/
numfunc = 400;
                        /*arbitrary number for example*/
/*compute first range buffer size*/
range = etext - *(uint *) start; /*init range*/
numtics = NUM HIST COUNTERS( range );
/*one counter for each 4 byte inst*/
num4fcnt = numfunc*sizeof( struct poutcnt )/HIST COUNTER SIZE;
BufferSize1 = numtics + num4fcnt;
/*compute second range buffer size*/
range = libhi-liblo;
BufferSize2 = range / 12; /*counter for every 12 inst bytes for\
a change*/
/*allocate buffer space - note: must be single contiguous\
```

```
buffer*/
pb[0].p buff = (HISTCOUNTER *)malloc( (BufferSize1 +BufferSize2)\
 *HIST COUNTER_SIZE);
if ( pb[0].p\_buff == NULL ) /*didn't get space - do error\
 recovery here* ;/
    return(-1);
/*set up the first range values*/
                                 /*start of main module*/
pb[0].p low = *(uint*)start;
pb[0].p_high = (caddr_t)etext;
                                  /*end of main module*/
pb[0].p_BufferSize = BufferSize1; /*prog addr cnt space + \
func cnt space*/
/*set up last element marker*/
pb[2].p high = (caddr t)0;
_mondata.prof_type = _PROF_TYPE_IS_P; /*define -p\
profiling*/
rc = monitor( (caddr t)1, (caddr t)1, pb, -1, numfunc); \
 /*start*/
if ( rc != 0 ) /*profiling did not start - do error recovery\
here*/
   return (-1);
/*other code for analysis ...*/
rc = monitor( (caddr_t)0); /*stop profiling and write data \
file mon.out*/
if ( rc != 0 )
                /*did not stop correctly - do error recovery\
 here*/
    return (-1);
```

3. This example shows how to profile contiguously loaded functions beginning at zit up to but not including zot with **-pg** profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
extern zit();
                       /*first function to profile*/
                       /*upper bound function*/
extern zot();
extern struct monglobal mondata; /*profiling global variables*/
                       /*monstartup return code*/
int rc;
_mondata.prof_type = _PROF_TYPE_IS_PG; /*define -pg profiling*/
/*Note cast used to obtain function code addresses*/
rc = monstartup(*(uint *)zit,*(uint *)zot); /*start*/
if ( rc != 0 ) /*profiling did not start, do error recovery\
here*/
   return(-1);
/*other code for analysis ...*/
           /*stop profiling and write data file gmon.out*/
```

Files

Data file for -p profiling. mon.out gmon.out Data file for -pg profiling.

Defines the _mondata.prof_type global variable in the monglobal data structure, /usr/include/mon.h

the **prof** structure, and the functions referred to in the previous examples.

Related Information

The moncontrol ("moncontrol Subroutine" on page 696) subroutine, monstartup ("monstartup Subroutine" on page 703) subroutine, **profil** ("profil Subroutine" on page 876) subroutine.

The **gprof** command, **prof** command.

The _end,_etext, or _edata ("_end, _etext, or _edata Identifier" on page 184) Identifier.

List of Memory Manipulation Services in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

monstartup Subroutine

Purpose

Starts and stops execution profiling using default-sized data areas.

Library

Standard C Library (libc.a)

Syntax

```
#include <mon.h>
int monstartup ( LowProgramCounter, HighProgramCounter)
0R
int monstartup((caddr t)-1), (caddr t) FragBuffer)
0R
int monstartup((caddr t)-1, (caddr t)0)
caddr t LowProgramCounter;
caddr_t HighProgramCounter;
```

Description

The monstartup subroutine allocates data areas of default size and starts profiling. Profiling causes periodic sampling and recording of the program location within the program address ranges specified, and accumulation of function-call count data for functions that have been compiled with the -p or -pq option.

Executable programs created with the cc -p or cc -pg command automatically include a call to the monstartup subroutine to profile the complete user program, including system libraries. In this case, you do not need to call the **monstartup** subroutine.

The monstartup subroutine is called by the mcrt0.o (-p) file or the gcrt0.o (-pg) file to begin profiling. The monstartup subroutine requires a global data variable to define whether -p or -pg profiling is to be in effect. The monstartup subroutine calls the monitor subroutine to initialize the data areas and start profiling.

The **prof** command is used to process the data file produced by -p profiling. The **gprof** command is used to process the data file produced by -pg profiling.

The monstartup subroutine examines the global and parameter data in the following order:

- 1. When the **mondata.prof** type global variable is neither -1 (-p profiling defined) nor +1 (-pg profiling defined), an error is returned and the function is considered complete.
 - The global variable is set to -1 in the mcrt0.o file and to +1 in the gcrt0.o file, and defaults to 0 when crt0.o is used.
- 2. When the LowProgramCounter value is not -1:
 - A single program address range is defined for profiling

AND

The first monstartup definition in the syntax is used to define the program range.

- 3. When the LowProgramCounter value is -1 and the HighProgramCounter value is not 0:
 - Multiple program address ranges are defined for profiling

AND

- The second monstartup definition in the syntax is used to define multiple ranges. The HighProgramCounter parameter, in this case, is the address of a frag structure array. The frag array size is denoted by a zero value for the HighProgramCounter (p high) field of the last element of the array. Each array element except the last defines one programming address range to be profiled. Programming ranges must be in ascending order of the program addresses with ascending order of the **prof** array index. Program ranges may not overlap.
- 4. When the LowProgramCounter value is -1 and the HighProgramCounter value is 0:
 - The whole program is defined for profiling

AND

 The third monstartup definition in the syntax is used. The program ranges are determined by monstartup and may be single range or multirange.

Parameters

LowProgramCounter (frag name: p low)

HighProgramCounter(frag name: p high)

Defines the lowest execution-time program address in the range to be profiled.

Defines the next address after the highest execution-time program address in the range to be profiled.

The program address parameters may be defined by function names or address expressions. If defined by a function name, then a function name expression must be used to dereference the function pointer to get the address of the first instruction in the function. This is required because the function reference in this context produces the address of the function descriptor. The first field of the descriptor is the address of the function code. See the examples for typical expressions to use. Specifies the address of a frag structure array.

FragBuffer

Examples

1. This example shows how to profile the main load module of a program with -p profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
extern caddr t etext;
                       /*system end of text
symbol*/
                       /*first function in main\
extern int start();
           program*/
extern struct monglobal mondata; /*profiling global variables*/
                  /*function
struct desc {
descriptor fields*/
  caddr t begin;
                    /*initial code
address*/
   caddr t toc; /*table of contents
address*/
                  /*environment
  caddr_t env;
pointer*/
}
            /*function
descriptor structure*/
struct desc *fd; /*pointer to function\
            descriptor*/
```

```
int rc;
                    /*monstartup
   return code*/
   fd = (struct desc *)start; /*init descriptor pointer to\
                start
   function*/
   mondata.prof type = PROF TYPE IS P; /*define -p profiling*/
   rc = monstartup( fd->begin, (caddr t) &etext); /*start*/
   if ( rc != 0 )
                        /*profiling did
   not start - do\
               error
   recovery here*/ return(-1);
               /*other code
   for analysis ...*/
   return(0);
                      /*stop profiling and
   write data\
                file
   mon.out*/
2. This example shows how to profile the complete program with -p profiling:
   #include <sys/types.h>
   #include <mon.h>
   main()
   extern struct monglobal _mondata; /*profiling global\
                  variables*/
   int rc;
                   /*monstartup
   return code*/
   mondata.prof type = PROF TYPE IS P; /*define -p profiling*/
   rc = monstartup( (caddr_t)-1, (caddr_t)0); /*start*/
   if ( rc != 0 )
                    /*profiling did
   not start -\
     do error recovery here*/
      return (-1);
               /*other code
   for analysis ...*/
                     /*stop profiling and
   return(0);
   write data\
                file
   mon.out*/
3. This example shows how to profile contiguously loaded functions beginning at zit up to but not
   including zot with -pg profiling:
   #include <sys/types.h>
   #include <mon.h>
   main()
   extern zit();
                        /*first function
   to profile*/
   extern zot();
                         /*upper bound
   function*/
   extern struct monglobal mondata; /*profiling global variables*/
   int rc;
                   /*monstartup
   return code*/
    mondata.prof type = PROF TYPE IS PG; /*define -pg profiling*/
   /*Note cast used to obtain function code addresses*/
   rc = monstartup(*(uint *)zit,*(uint *)zot); /*start*/
   if ( rc != 0 )
                         /*profiling did
   not start - do\
                error
```

recovery here*/ return(-1);

```
/*other code for analysis ...*/ exit(0); /*stop profiling and write data file gmon.out*/ }
```

Return Values

The **monstartup** subroutine returns 0 upon successful completion.

Error Codes

If an error is found, the monstartup subroutine outputs an error message to stderr and returns -1.

Files

mon.out Data file for -p profiling.
gmon.out Data file for -pg profiling.

mon.h Defines the mondata.prof type variable in the monglobal data structure, the prof structure, and

the functions referred to in the examples.

Related Information

The **moncontrol** ("moncontrol Subroutine" on page 696) subroutine, **monitor** ("monitor Subroutine" on page 697) subroutine, **profil** ("profil Subroutine" on page 876) subroutine.

The **gprof** command, **prof** command.

The _end, _etext, or _edata ("_end, _etext, or _edata Identifier" on page 184) Identifier.

List of Memory Manipulation Services in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

mprotect Subroutine

Purpose

Modifies access protections for memory mapping.

Library

Standard C Library (libc.a)

Syntax

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

int mprotect ( addr, len, prot)
void *addr;
size_t len;
int prot;
```

Description

The **mprotect** subroutine modifies the access protection of a mapped file region or anonymous memory region created by the **mmap** subroutine. The behavior of this function is unspecified if the mapping was not established by a call to the **mmap** subroutine.

Parameters

addr Specifies the address of the region to be modified. Must be a multiple of the page size returned by the sysconf subroutine using the SC PAGE SIZE value for the Name parameter.

len Specifies the length, in bytes, of the region to be modified. If the len parameter is not a multiple of the page size returned by the sysconf subroutine using the SC PAGE SIZE value for the Name parameter, the length of the region will be rounded off to the next multiple of the page size.

prot Specifies the new access permissions for the mapped region. Legitimate values for the prot parameter are the same as those permitted for the **mmap** ("mmap or mmap64 Subroutine" on page 690) subroutine, as follows:

PROT READ

Region can be read.

PROT WRITE

Region can be written.

PROT_EXEC

Region can be executed.

PROT NONE

Region cannot be accessed.

Return Values

When successful, the mprotect subroutine returns 0. Otherwise, it returns -1 and sets the errno global variable to indicate the error.

Note: The return value for mprotect is 0 if it fails because the region given was not created by mmap unless XPG 1170 behavior is requested by setting the environment variable XPG SUS ENV to ON.

Error Codes

Attention: If the mprotect subroutine is unsuccessful because of a condition other than that specified by the **EINVAL** error code, the access protection for some pages in the (addr, addr + len) range may have been changed.

If the **mprotect** subroutine is unsuccessful, the **errno** global variable may be set to one of the following values:

EACCES The prot parameter specifies a protection that conflicts with the access permission set for the

underlying file.

EINVAL The prot parameter is not valid, or the addr parameter is not a multiple of the page size as

> returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter. The application has requested Single UNIX Specification, Version 2 compliant behavior and

> addresses in the range are invalid for the address space of the process or specify one or more

pages which are not mapped.

msem init Subroutine

Purpose

ENOMEM

Initializes a semaphore in a mapped file or shared memory region.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/mman.h>
msemaphore *msem init ( Sem, InitialValue)
msemaphore *Sem;
int Initial Value:
```

Description

The **msem** init subroutine allocates a new binary semaphore and initializes the state of the new semaphore.

If the value of the InitialValue parameter is MSEM_LOCKED, the new semaphore is initialized in the locked state. If the value of the InitialValue parameter is MSEM_UNLOCKED, the new semaphore is initialized in the unlocked state.

The **msemaphore** structure is located within a mapped file or shared memory region created by a successful call to the mmap subroutine and having both read and write access.

Whether a semaphore is created in a mapped file or in an anonymous shared memory region, any reference by a process that has mapped the same file or shared region, using an msemaphore structure pointer that resolved to the same file or start of region offset, is taken as a reference to the same semaphore.

Any previous semaphore state stored in the **msemaphore** structure is ignored and overwritten.

Parameters

Sem Points to an msemaphore structure in which the state of the semaphore is stored.

Initial Value Determines whether the semaphore is locked or unlocked at allocation.

Return Values

When successful, the **msem** init subroutine returns a pointer to the initialized **msemaphore** structure. Otherwise, it returns a null value and sets the **errno** global variable to indicate the error.

Error Codes

If the msem_init subroutine is unsuccessful, the errno global variable is set to one of the following values:

Indicates the InitialValue parameter is not valid. ENOMEM Indicates a new semaphore could not be created.

Related Information

The mmap ("mmap or mmap64 Subroutine" on page 690) subroutine, msem lock ("msem lock Subroutine" on page 709) subroutine, **msem remove** ("msem remove Subroutine" on page 710) subroutine, msem_unlock ("msem_unlock Subroutine" on page 711) subroutine.

List of Memory Mapping Services and Understanding Memory Mapping in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

msem_lock Subroutine

Purpose

Locks a semaphore.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/mman.h>
int msem lock ( Sem, Condition)
msemaphore *Sem;
int Condition;
```

Description

The **msem_lock** subroutine attempts to lock a binary semaphore.

If the semaphore is not currently locked, it is locked and the **msem lock** subroutine completes successfully.

If the semaphore is currently locked, and the value of the Condition parameter is MSEM_IF_NOWAIT, the msem lock subroutine returns with an error. If the semaphore is currently locked, and the value of the Condition parameter is 0, the **msem_lock** subroutine does not return until either the calling process is able to successfully lock the semaphore or an error condition occurs.

All calls to the msem_lock and msem_unlock subroutines by multiple processes sharing a common msemaphore structure behave as if the call were serialized.

If the **msemaphore** structure contains any value not resulting from a call to the **msem init** subroutine, followed by a (possibly empty) sequence of calls to the msem_lock and msem_unlock subroutines, the results are undefined. The address of an **msemaphore** structure is significant. If the **msemaphore** structure contains any value copied from an msemaphore structure at a different address, the result is undefined.

Parameters

Sem Points to an msemaphore structure that specifies the semaphore to be locked.

Condition Determines whether the msem_lock subroutine waits for a currently locked semaphore to unlock.

Return Values

When successful, the msem_lock subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the errno global variable to indicate the error.

Error Codes

If the msem_lock subroutine is unsuccessful, the errno global variable is set to one of the following values:

EAGAIN Indicates a value of MSEM_IF_NOWAIT is specified for the Condition parameter and the semaphore is already locked.

EINVAL Indicates the Sem parameter points to an msemaphore structure specifying a semaphore that has been

removed, or the Condition parameter is invalid.

EINTR Indicates the **msem_lock** subroutine was interrupted by a signal that was caught.

Related Information

The msem_init ("msem_init Subroutine" on page 707) subroutine, msem_remove ("msem_remove Subroutine") subroutine, msem_unlock ("msem_unlock Subroutine" on page 711) subroutine.

List of Memory Mapping Services and Understanding Memory Mapping in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

msem remove Subroutine

Purpose

Removes a semaphore.

Library

Standard C Library (libc.a)

Syntax

#include <sys/mman.h>

int msem remove (Sem) msemaphore *Sem;

Description

The **msem** remove subroutine removes a binary semaphore. Any subsequent use of the **msemaphore** structure before it is again initialized by calling the msem_init subroutine will have undefined results.

The msem_remove subroutine also causes any process waiting in the msem_lock subroutine on the removed semaphore to return with an error.

If the **msemaphore** structure contains any value not resulting from a call to the **msem init** subroutine, followed by a (possibly empty) sequence of calls to the msem lock and msem unlock subroutines, the result is undefined. The address of an msemaphore structure is significant. If the msemaphore structure contains any value copied from an msemaphore structure at a different address, the result is undefined.

Parameters

Sem Points to an **msemaphore** structure that specifies the semaphore to be removed.

Return Values

When successful, the msem_remove subroutine returns a value of 0. Otherwise, it returns a -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the msem_remove subroutine is unsuccessful, the errno global variable is set to the following value:

EINVAL Indicates the Sem parameter points to an msemaphore structure that specifies a semaphore that has been removed.

Related Information

The msem_init ("msem_init Subroutine" on page 707) subroutine, msem_lock ("msem_lock Subroutine" on page 709) subroutine, msem_unlock ("msem_unlock Subroutine") subroutine.

List of Memory Mapping Services and Understanding Memory Mapping in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

msem unlock Subroutine

Purpose

Unlocks a semaphore.

Library

Standard C Library (libc.a)

Syntax

#include <sys/mman.h>

int msem unlock (Sem, Condition) msemaphore *Sem; int Condition;

Description

The **msem_unlock** subroutine attempts to unlock a binary semaphore.

If the semaphore is currently locked, it is unlocked and the msem_unlock subroutine completes successfully.

If the Condition parameter is 0, the semaphore is unlocked, regardless of whether or not any other processes are currently attempting to lock it. If the Condition parameter is set to the MSEM IF WAITERS value, and another process is waiting to lock the semaphore or it cannot be reliably determined whether some process is waiting to lock the semaphore, the semaphore is unlocked by the calling process. If the Condition parameter is set to the MSEM IF WAITERS value and no process is waiting to lock the semaphore, the semaphore will not be unlocked and an error will be returned.

Parameters

Sem Points to an **msemaphore** structure that specifies the semaphore to be unlocked.

Determines whether the msem_unlock subroutine unlocks the semaphore if no other processes Condition

are waiting to lock it.

Return Values

When successful, the msem_unlock subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the errno global variable to indicate the error.

Error Codes

If the msem_unlock subroutine is unsuccessful, the errno global variable is set to one of the following values:

EAGAIN **EINVAL**

Indicates a Condition value of MSEM_IF_WAITERS was specified and there were no waiters. Indicates the Sem parameter points to an msemaphore structure specifying a semaphore that has been removed, or the Condition parameter is not valid.

Related Information

The msem_init ("msem_init Subroutine" on page 707) subroutine, msem_lock ("msem_lock Subroutine" on page 709) subroutine, msem_remove ("msem_remove Subroutine" on page 710) subroutine.

List of Memory Mapping Services and Understanding Memory Mapping in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

msgctl Subroutine

Purpose

Provides message control operations.

Library

Standard C Library (libc.a)

Syntax

#include <sys/msg.h>

int msgctl (MessageQueueID,Command,Buffer) int MessageQueueID, Command; struct msqid ds * Buffer;

Description

The msgctl subroutine provides a variety of message control operations as specified by the Command parameter and stored in the structure pointed to by the Buffer parameter. The msqid_ds structure is defined in the sys/msg.h file.

The following limits apply to the message queue:

- Maximum message size is 65,535 bytes for releases prior to AIX 4.1.5 and is 4 Megabytes for release AIX 4.1.5 and later releases.
- Maximum number of messages per queue is 8192.
- Maximum number of message queue IDs is 4096 for releases before AIX 4.3.2 and 131072 for AIX 4.3.2 and following.
- Maximum number of bytes in a queue is 4 65,535 for releases prior to AIX 4.1.5 and is 4 Megabytes for release 4.1.5 and later releases.

Parameters

MessageQueueID

Specifies the message queue identifier.

Command

The following values for the Command parameter are available:

IPC_STAT

Stores the current value of the above fields of the data structure associated with the MessageQueueID parameter into the msqid_ds structure pointed to by the Buffer parameter.

The current process must have read permission in order to perform this operation.

IPC_SET

Sets the value of the following fields of the data structure associated with the MessageQueueID parameter to the corresponding values found in the structure pointed to by the Buffer parameter:

```
msg perm.uid
msg_perm.gid
msg perm.mode/*Only the low-order
nine bits*/
msg qbytes
```

The effective user ID of the current process must have root user authority or must equal the value of the msg perm.uid or msg perm.cuid field in the data structure associated with the MessageQueueID parameter in order to perform this operation. To raise the value of the msg qbytes field, the effective user ID of the current process must have root user authority.

IPC RMID

Removes the message queue identifier specified by the MessageQueueID parameter from the system and destroys the message queue and data structure associated with it. The effective user ID of the current process must have root user authority or be equal to the value of the msg perm.uid or msg perm.cuid field in the data structure associated with the MessageQueueID parameter to perform this operation.

Buffer

Points to a **msqid** ds structure.

Return Values

Upon successful completion, the msgctl subroutine returns a value of 0. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **msgctl** subroutine is unsuccessful if any of the following conditions is true:

EINVAL The Command or MessageQueueID parameter is not valid.

EACCES The Command parameter is equal to the IPC_STAT value, and the calling process was denied read

permission.

EPERM The Command parameter is equal to the IPC_RMID value and the effective user ID of the calling process

does not have root user authority. Or, the Command parameter is equal to the IPC_SET value, and the effective user ID of the calling process is not equal to the value of the msg perm.uid field or the msg perm.cuid field in the data structure associated with the MessageQueuelD parameter.

EPERM The Command parameter is equal to the IPC_SET value, an attempt was made to increase the value of

the msg qbytes field, and the effective user ID of the calling process does not have root user authority.

EFAULT The Buffer parameter points outside of the process address space.

Related Information

The **msqqet** ("msqqet Subroutine" on page 714) subroutine, **msqrcv** ("msqrcv Subroutine" on page 715) subroutine, msgsnd ("msgsnd Subroutine" on page 718) subroutine, msgxrcv ("msgxrcv Subroutine" on page 720) subroutine.

msgget Subroutine

Purpose

Gets a message queue identifier.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/msg.h>
int msgget ( Key, MessageFlag)
key t Key;
int MessageFlag;
```

Description

The **msgget** subroutine returns the message queue identifier associated with the specified *Key* parameter.

A message queue identifier, associated message queue, and data structure are created for the value of the *Key* parameter if one of the following conditions is true:

- The Key parameter is equal to the IPC_PRIVATE value.
- The Key parameter does not already have a message queue identifier associated with it, and the **IPC_CREAT** value is set.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- The msg perm.cuid, msg perm.uid, msg perm.cgid, and msg perm.gid fields are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of the msg perm.mode field are set equal to the low-order 9 bits of the MessageFlag parameter.
- The msg qnum, msg lspid, msg lrpid, msg stime, and msg rtime fields are set equal to 0.
- The msg ctime field is set equal to the current time.
- · The msg qbytes field is set equal to the system limit.

The **msgget** subroutine performs the following actions:

- The **msgget** subroutine either finds or creates (depending on the value of the *MessageFlag* parameter) a queue with the Key parameter.
- · The msgget subroutine returns the ID of the queue header to its caller.

Limits on message size and number of messages in the queue can be found in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Parameters

Key

Specifies either the value IPC_PRIVATE or an Interprocess Communication (IPC) key constructed by the ftok ("ftok Subroutine" on page 277) subroutine (or by a similar algorithm).

MessageFlag

Constructed by logically ORing one or more of the following values:

IPC CREAT

Creates the data structure if it does not already exist.

IPC EXCL

Causes the msgget subroutine to fail if the IPC_CREAT value is also set and the data structure already exists.

S IRUSR

Permits the process that owns the data structure to read it.

S IWUSR

Permits the process that owns the data structure to modify it.

S IRGRP

Permits the group associated with the data structure to read it.

S IWGRP

Permits the group associated with the data structure to modify it.

S IROTH

Permits others to read the data structure.

S_IWOTH

Permits others to modify the data structure.

Values that begin with S I are defined in the sys/mode.h file and are a subset of the access permissions that apply to files.

Return Values

Upon successful completion, the msgget subroutine returns a message queue identifier. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **msgget** subroutine is unsuccessful if any of the following conditions is true:

EACCES	A message queue identifier exists for the <i>Key</i> parameter, but operation permission as specified by the
	less and a O lite of the Manager Florida manager to it and amounted

low-order 9 bits of the *MessageFlag* parameter is not granted.

ENOENT A message queue identifier does not exist for the Key parameter and the IPC_CREAT value is not set. **ENOSPC** A message queue identifier is to be created, but the system-imposed limit on the maximum number of

allowed message queue identifiers system-wide would be exceeded.

EEXIST A message queue identifier exists for the Key parameter, and both IPC_CREAT and IPC_EXCL values

are set.

Related Information

The **ftok** ("ftok Subroutine" on page 277) subroutine, **msgctl** ("msgctl Subroutine" on page 712) subroutine, msgrcv ("msgrcv Subroutine") subroutine, msgsnd ("msgsnd Subroutine" on page 718) subroutine, msgxrcv ("msgxrcv Subroutine" on page 720) subroutine.

The mode.h file.

msgrcv Subroutine

Purpose

Reads a message from a queue.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/msg.h>
int msgrcv (MessageQueueID, MessagePointer,MessageSize,MessageType, MessageFlag)
int MessageQueueID, MessageFlag;
void * MessagePointer;
size t MessageSize;
long int MessageType;
```

Description

The **msgrcv** subroutine reads a message from the queue specified by the *MessageQueueID* parameter and stores it into the structure pointed to by the MessagePointer parameter. The current process must have read permission in order to perform this operation.

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64-bit application calling 32-bit kernel interface.

Limits on message size and number of messages in the queue can be found in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Note: For a 64-bit process, the mtype field is 64 bits long. However, for compatibility with 32-bit processes, the mtype field must be a 32-bit signed value that is sign-extended to 64 bits. The most significant 32 bits are not put on the message queue. For a 64-bit process, the mtype field is again sign-extended to 64 bits.

Parameters

MessageQueueID MessagePointer

Specifies the message queue identifier.

Points to a msgbuf structure containing the message. The msgbuf structure is defined in the sys/msg.h file and contains the following fields:

```
/* Message type */
mtyp t
        mtype;
                      /* Beginning of message text */
char
        mtext[1];
```

The mtype field contains the type of the received message as specified by the sending process. The mtext field is the text of the message.

MessageSize

Specifies the size of the mtext field in bytes. The received message is truncated to the size specified by the MessageSize parameter if it is longer than the size specified by the MessageSize parameter and if the MSG_NOERROR value is set in the MessageFlag parameter. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

MessageType

Specifies the type of message requested as follows:

- If equal to the value of 0, the first message on the gueue is received.
- If greater than 0, the first message of the type specified by the MessageType parameter is received.
- · If less than 0, the first message of the lowest type that is less than or equal to the absolute value of the *MessageType* parameter is received.

MessageFlag

Specifies either a value of 0 or is constructed by logically ORing one or more of the following values:

MSG_NOERROR

Truncates the message if it is longer than the MessageSize parameter.

IPC NOWAIT

Specifies the action to take if a message of the desired type is not on the queue:

- If the IPC_NOWAIT value is set, the calling process returns a value of -1 and sets the errno global variable to the ENOMSG error code.
- · If the IPC NOWAIT value is not set, the calling process suspends execution until one of the following occurs:
 - A message of the desired type is placed on the queue.
 - The message queue identifier specified by the MessageQueueID parameter is removed from the system. When this occurs, the errno global variable is set to the EIDRM error code, and a value of -1 is returned.
 - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner described in the sigaction subroutine.

Return Values

Upon successful completion, the **msqrcv** subroutine returns a value equal to the number of bytes actually stored into the mtext field and the following actions are taken with respect to fields of the data structure associated with the MessageQueueID parameter:

- The msg qnum field is decremented by 1.
- The msg 1rpid field is set equal to the process ID of the calling process.
- The msg rtime field is set equal to the current time.

If the msgrcv subroutine is unsuccessful, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The msgrcv subroutine is unsuccessful if any of the following conditions is true:

EINVAL The MessageQueueID parameter is not a valid message queue identifier. **EACCES** The calling process is denied permission for the specified operation.

F2RIG The mtext field is greater than the MessageSize parameter, and the MSG_NOERROR value is not set.

ENOMSG The queue does not contain a message of the desired type and the IPC NOWAIT value is set. **EFAULT** The MessagePointer parameter points outside of the allocated address space of the process.

EINTR The **msgrcv** subroutine is interrupted by a signal.

EIDRM The message queue identifier specified by the MessageQueueID parameter has been removed from

the system.

Related Information

The **msgctl** ("msgctl Subroutine" on page 712) subroutine, **msgget** ("msgget Subroutine" on page 714) subroutine, msgsnd ("msgsnd Subroutine" on page 718) subroutine, msgxrcv ("msgxrcv Subroutine" on page 720) subroutine, sigaction subroutine.

msgsnd Subroutine

Purpose

Sends a message.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/msg.h>
int msgsnd (MessageQueueID, MessagePointer,MessageSize, MessageFlag)
int MessageQueueID, MessageFlag;
const void * MessagePointer;
size t MessageSize;
```

Description

The **msgsnd** subroutine sends a message to the queue specified by the *MessageQueueID* parameter. The current process must have write permission to perform this operation. The *MessagePointer* parameter points to an **msgbuf** structure containing the message. The **sys/msg.h** file defines the **msgbuf** structure. The structure contains the following fields:

```
mtyp_t mtype; /* Message type */
char mtext[1]; /* Beginning of message text */
```

The mtype field specifies a positive integer used by the receiving process for message selection. The mtext field can be any text of the length in bytes specified by the *MessageSize* parameter. The *MessageSize* parameter can range from 0 to the maximum limit imposed by the system.

The following example shows a typical user-defined **msgbuf** structure that includes sufficient space for the largest message:

```
struct my_msgbuf
mtyp_t     mtype;
char     mtext[MSGSIZ]; /* MSGSIZ is the size of the largest message */
```

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64-bit application calling 32-bit kernel interface.

The following system limits apply to the message queue:

- Maximum message size is 65,535 bytes for releases prior to AIX 4.1.5 and is 4 Megabytes for AIX 4.1.5 and later releases.
- · Maximum number of messages per queue is 8192.
- Maximum number of message queue IDs is 4096 for releases before AIX 4.3.2 and 131072 for AIX 4.3.2 and following.
- Maximum number of bytes in a queue is 4 65,535 bytes for releases prior to AIX 4.1.5 is 4 Megabytes for AIX 4.1.5 and later releases.

Note: For a 64-bit process, the **mtype** field is 64 bits long. However, for compatibility with 32-bit processes, the **mtype** field must be a 32-bit signed value that is sign-extended to 64 bits. The most significant 32 bits are not put on the message queue. For a 64-bit process, the **mtype** field is again sign-extended to 64 bits.

The MessageFlag parameter specifies the action to be taken if the message cannot be sent for one of the following reasons:

- The number of bytes already on the queue is equal to the number of bytes defined by themsg_qbytes structure.
- The total number of messages on the queue is equal to a system-imposed limit.

These actions are as follows:

- If the MessageFlag parameter is set to the IPC_NOWAIT value, the message is not sent, and the msgsnd subroutine returns a value of -1 and sets the errno global variable to the EAGAIN error code.
- If the MessageFlag parameter is set to 0, the calling process suspends execution until one of the following occurs:
 - The condition responsible for the suspension no longer exists, in which case the message is sent.
 - The MessageQueueID parameter is removed from the system. (For information on how to remove the MessageQueueID parameter, see the msgctl ("msgctl Subroutine" on page 712) subroutine.) When this occurs, the errno global variable is set equal to the EIDRM error code, and a value of -1
 - The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in the **sigaction** subroutine.

Parameters

MessageQueueID Specifies the queue to which the message is sent. Points to a **msqbuf** structure containing the message. MessagePointer MessageSize Specifies the length, in bytes, of the message text.

MessageFlag Specifies the action to be taken if the message cannot be sent.

Return Values

Upon successful completion, a value of 0 is returned and the following actions are taken with respect to the data structure associated with the MessageQueueID parameter:

- The msg_gnum field is incremented by 1.
- The msg 1spid field is set equal to the process ID of the calling process.
- The msg stime field is set equal to the current time.

If the msgsnd subroutine is unsuccessful, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **msgsnd** subroutine is unsuccessful and no message is sent if one or more of the following conditions is true:

EACCES The calling process is denied permission for the specified operation.

EAGAIN The message cannot be sent for one of the reasons stated previously, and the MessageFlag parameter

is set to the IPC_NOWAIT value or the system has temporarily ran out of memory resource.

EFAULT The MessagePointer parameter points outside of the address space of the process.

EIDRM The message queue identifier specified by the MessageQueueID parameter has been removed from the

system.

EINTR The **msgsnd** subroutine received a signal.

EINVAL The MessageQueueID parameter is not a valid message queue identifier.

EINVAL The mtype field is less than 1.

EINVAL The MessageSize parameter is less than 0 or greater than the system-imposed limit.

EINVAL The upper 32-bits of the 64-bit mtype field for a 64-bit process is not 0.

ENOMEM The message could not be sent because not enough storage space was available.

Related Information

The **msgctl** ("msgctl Subroutine" on page 712) subroutine, **msgget** ("msgget Subroutine" on page 714) subroutine, **msgrcv** ("msgrcv Subroutine" on page 715) subroutine, **msgrcv** ("msgxrcv Subroutine") subroutine, **sigaction** subroutine.

msgxrcv Subroutine

Purpose

Receives an extended message.

Library

Standard C Library (libc.a)

Syntax

```
For releases prior to AIX 4.3:
#include <sys/msg.h>

int msgxrcv (MessageQueueID, MessagePointer, MessageSize, MessageType, MessageFlag)
int MessageQueueID, MessageFlag, MessageSize;
struct msgxbuf * MessagePointer;
long MessageType;

For AIX 4.3 and later releases:
#include <sys/msg.h>

int msgxrcv (MessageQueueID, MessagePointer, MessageSize, MessageType, MessageFlag)
int MessageQueueID, MessageFlag;
size_t MessageSize;
struct msgxbuf * MessagePointer;
long MessageType;
```

Description

The **msgxrcv** subroutine reads a message from the queue specified by the *MessageQueueID* parameter and stores it into the extended message receive buffer pointed to by the *MessagePointer* parameter. The current process must have read permission in order to perform this operation. The **msgxbuf** structure is defined in the **sys/msg.h** file.

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64-bit application calling 32-bit kernel interface.

The following limits apply to the message queue:

- Maximum message size is 65,535 bytes for releases prior to AIX 4.1.5 and is 4 Megabytes for AIX 4.1.5 and later releases.
- Maximum number of messages per queue is 8192.
- Maximum number of message queue IDs is 4096 for releases before AIX 4.3.2 and 131072 for AIX 4.3.2 and following.
- Maximum number of bytes in a queue is 4 65,535 for releases prior to AIX 4.1.5 and is 4 Megabytes for AIX 4.1.5 later releases.

Note: For a 64-bit process, the mtype field is 64 bits long. However, for compatibility with 32-bit processes, the mtype field must be a 32-bit signed value that is sign-extended to 64 bits. The most significant 32 bits are not put on the message queue. For a 64-bit process, the mtype field is again sign-extended to 64 bits.

Parameters

MessageQueueID MessagePointer MessageSize

Specifies the message queue identifier.

Specifies a pointer to an extended message receive buffer where a message is stored. Specifies the size of the mtext field in bytes. The receive message is truncated to the size specified by the MessageSize parameter if it is larger than the MessageSize parameter and the MSG_NOERROR value is true. The truncated part of the message is lost and no indication of the truncation is given to the calling process. If the message is longer than the number of bytes specified by the MessageSize parameter and the MSG NOERROR value is not set, the msgxrcv subroutine is unsuccessful and sets the errno global variable to the E2BIG error code.

MessageType

Specifies the type of message requested as follows:

- If the MessageType parameter is equal to 0, the first message on the queue is received.
- If the MessageType parameter is greater than 0, the first message of the type specified by the *MessageType* parameter is received.
- If the MessageType parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the MessageType parameter is received.

Specifies a value of 0 or a value constructed by logically ORing one or more of the following values:

MessageFlag

MSG NOERROR

Truncates the message if it is longer than the number of bytes specified by the MessageSize parameter.

IPC_NOWAIT

Specifies the action to take if a message of the desired type is not on the queue:

- If the IPC_NOWAIT value is set, the calling process returns a value of -1 and sets the errno global variable to the ENOMSG error code.
- If the IPC_NOWAIT value is not set, the calling process suspends execution until one of the following occurs:
 - A message of the desired type is placed on the queue.
 - The message queue identifier specified by the MessageQueueID parameter is removed from the system. When this occurs, the errno global variable is set to the **EIDRM** error code, and a value of -1 is returned.
 - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner prescribed in the sigaction subroutine.

Return Values

Upon successful completion, the **msqxrcv** subroutine returns a value equal to the number of bytes actually stored into the mtext field, and the following actions are taken with respect to the data structure associated with the MessageQueueID parameter:

- The msg qnum field is decremented by 1.
- The msg lrpid field is set equal to the process ID of the calling process.
- The msg rtime field is set equal to the current time.

If the msqxrcv subroutine is unsuccessful, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The msgxrcv subroutine is unsuccessful if any of the following conditions is true:

EINVAL The MessageQueueID parameter is not a valid message queue identifier. **EACCES** The calling process is denied permission for the specified operation.

EINVAL The MessageSize parameter is less than 0.

E2BIG The mtext field is greater than the MessageSize parameter, and the MSG_NOERROR value is not set.

ENOMSG The queue does not contain a message of the desired type and the IPC NOWAIT value is set.

EFAULT The MessagePointer parameter points outside of the process address space.

EINTR The **msgxrcv** subroutine was interrupted by a signal.

EIDRM The message queue identifier specified by the MessageQueueID parameter is removed from the system.

Related Information

The msgctl ("msgctl Subroutine" on page 712) subroutine, msgget ("msgget Subroutine" on page 714) subroutine, msqrcv ("msqrcv Subroutine" on page 715) subroutine, msqsnd ("msqsnd Subroutine" on page 718) subroutine, sigaction subroutine.

msleep Subroutine

Purpose

Puts a process to sleep when a semaphore is busy.

Library

Standard C Library (libc.a)

Syntax

#include <sys/mman.h>

int msleep (Sem) msemaphore * Sem;

Description

The msleep subroutine puts a calling process to sleep when a semaphore is busy. The semaphore should be located in a shared memory region. Use the **mmap** subroutine to create the shared memory section.

All of the values in the **msemaphore** structure must result from a **msem_init** subroutine call. This call may or may not be followed by a sequence of calls to the msem lock subroutine or the msem unlock subroutine. If the **msemaphore** structure value originates in another manner, the results of the **msleep** subroutine are undefined.

The address of the **msemaphore** structure is significant. You should be careful not to modify the structure's address. If the structure contains values copied from a msemaphore structure at another address, the results of the **msleep** subroutine are undefined.

Parameters

Sem Points to the **msemaphore** structure that specifies the semaphore.

Error Codes

If the msleep subroutine is unsuccessful, the errno global variable is set to one of the following values:

EFAULT Indicates that the Sem parameter points to an invalid address or the address does not contain a valid

msemaphore structure.

EINTR Indicates that the process calling the msleep subroutine was interrupted by a signal while sleeping.

Related Information

The mmap ("mmap or mmap64 Subroutine" on page 690) subroutine, msem_init ("msem_init Subroutine" on page 707) subroutine, **msem lock** ("msem lock Subroutine" on page 709) subroutine, **msem unlock** ("msem_unlock Subroutine" on page 711) subroutine, mwakeup ("mwakeup Subroutine" on page 727) subroutine.

Understanding Memory Mapping in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

msync Subroutine

Purpose

Synchronizes a mapped file.

Library

Standard C Library (libc.a).

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>
int msync (addr, len, flags)
void *addr;
size t len;
int flags;
```

Description

The **msync** subroutine controls the caching operations of a mapped file region. Use the **msync** subroutine to transfer modified pages in the region to the underlying file storage device.

If the application has requested Single UNIX Specification, Version 2 compliant behavior then the st_ctime and st_mtime fields of the mapped file are marked for update upon successful completion of the msync subroutine call if the file has been modified.

Parameters

Specifies the address of the region to be synchronized. Must be a multiple of the page size returned by the addr sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter.

Specifies the length, in bytes, of the region to be synchronized. If the len parameter is not a multiple of the len page size returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter, the length of the region is rounded up to the next multiple of the page size.

flags

Specifies one or more of the following symbolic constants that determine the way caching operations are performed:

MS_SYNC

Specifies synchronous cache flush. The **msync** subroutine does not return until the system completes all I/O operations.

This flag is invalid when the MAP PRIVATE flag is used with the mmap subroutine.

MAP_PRIVATE is the default privacy setting. When the MS_SYNC and MAP_PRIVATE flags both are used, the msync subroutine returns an errno value of EINVAL.

MS_ASYNC

Specifies an asynchronous cache flush. The **msync** subroutine returns after the system schedules all I/O operations.

This flag is invalid when the **MAP_PRIVATE** flag is used with the **mmap** subroutine.

MAP_PRIVATE is the default privacy setting. When the MS_SYNC and MAP_PRIVATE flags both are used, the msync subroutine returns an errno value of EINVAL.

MS INVALIDATE

Specifies that the **msync** subroutine invalidates all cached copies of the pages. New copies of the pages must then be obtained from the file system the next time they are referenced.

Return Values

When successful, the **msync** subroutine returns 0. Otherwise, it returns -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the msync subroutine is unsuccessful, the errno global variable is set to one of the following values:

EIO An I/O error occurred while reading from or writing to the file system.

ENOMEM The range specified by (addr, addr + len) is invalid for a process' address space, or the range specifies

one or more unmapped pages.

EINVAL The addr argument is not a multiple of the page size as returned by the **sysconf** subroutine using the

_SC_PAGE_SIZE value for the *Name* parameter, or the *flags* parameter is invalid. The address of the

region is within the process' inheritable address space.

mt trce Subroutine

Purpose

Dumps traceback information into a lightweight core file.

Library

PTools Library (libptools_ptr.a)

Syntax

void mt trce (int FileDescriptor, int Signal, struct sigcontext *Context, int Node);

Description

The **mt_trce** subroutine dumps traceback information of the calling thread and all other threads allocated in the process space into the file specified by the *FileDescriptor* parameter. The format of the output from this subroutine complies with the Parallel Tools Consortium Lightweight CoreFile Format. Threads, except the calling thread, will be suspended after the calling thread enters this subroutine and while the traceback information is being obtained. Threads execution resumes when this subroutine returns.

When using the **mt trce** subroutine in a signal handler, it is recommended that the application be started with the environment variable AIXTHREAD SCOPE set to S (As in export AIXTHREAD SCOPE=S). If this variable is not set, the application may hang.

Parameters

Context Points to the sigcontext structure containing the context of the thread when the signal

> happens. The context is used to generate the traceback information for the calling thread. This is used only if the Signal parameter is nonzero. If the mt trce subroutine is called with the Signal parameter set to zero, the Context parameter is ignored and the traceback information is generated based on the current context of the calling thread. Refer to the sigaction subroutine for further description about signal handlers and how the sigcontext structure is

passed to a signal handler.

File Descriptor The file descriptor of the lightweight core file. It specifies the target file into which the

traceback information is written.

Specifies the number of the tasks or nodes where this subroutine is executing and is used only Node

> for a parallel application consisting of multiple tasks. The Node parameter will be used in section headers of the traceback information to identify the task or node from which the

information is generated.

The number of the signal that causes the signal handler to be executed. This is used only if Signal

> the mt_trce subroutine is called from a signal handler. A Fault-Info section defined by the Parallel Tools Consortium Lightweight Core File Format will be written into the output lightweight core file based on this signal number. If the mt_trce subroutine is not called from a signal handler, the Signal parameter must be set to 0 and a Fault-Info section will not be

generated.

Notes:

- 1. To obtain source line information in the traceback, the programs must have been compiled with the -g option to include the necessary line number information in the executable files. Otherwise, address offset from the beginning of the function is provided.
- 2. Line number information is not provided for shared objects even if they were compiled with the -q option.
- 3. Function names are not provided if a program or a library is compiled with optimization. To obtain function name information in the traceback and still have the object code optimized, compiler option -atbtable=full must be specified.
- 4. In rare cases, the traceback of a thread may seem to skip one level of procedure calls. This is because the traceback is obtained at the moment the thread entered a procedure and has not vet allocated a stack frame.

Return Values

Upon successful completion, the mt_trce subroutine returns a value of 0. Otherwise, an error number is returned to indicate the error.

Error Codes

If an error occurs, the subroutine returns -1 and the error global variable is set to indicate the error, as follows:

EBADF The FileDescriptor parameter does not specify a valid file descriptor open for writing.

ENOSPC No free space is left in the file system containing the file.

EDQUOT New disk blocks cannot be allocated for the file because the user or group quota of blocks has

been exhausted on the file system.

EINVAL The value of the Signal parameter is invalid or the Context parameter points to an invalid

context.

ENOMEM Insufficient memory exists to perform the operation.

Examples

1. The following example calls the **mt__trce** subroutine to generate traceback information in a signal handler.

The following is an example of the lightweight core file generated by the mt_trce subroutine. Notice the thread ID in the information is the unique sequence number of a thread for the life time of the process containing the thread.

```
+++PARALLEL TOOLS CONSORTIUM LIGHTWEIGHT COREFILE FORMAT version 1.0
+++LCB 1.0 Thu Jun 30 16:02:35 1999 Generated by AIX
+++ID Node 0 Process 21084 Thread 1
***FAULT "SIGABRT - Abort"
+++STACK
func2 : 123 # in file
func1 : 272 # in file
main: 49 # in file
---STACK
---ID Node 0 Process 21084 Thread 1
+++ID Node 0 Process 21084 Thread 2
+++STACK
nsleep : 0x0000001c
sleep: 0x00000030
f mt exec : 21 # in file
_pthread_body : 0x00000114
---STACK
---ID Node 0 Process 21084 Thread 2
+++ID Node 0 Process 21084 Thread 3
+++STACK
nsleep : 0x0000001c
sleep : 0x00000030
f mt exec : 21 # in file
_pthread_body : 0x00000114
---STACK
---ID Node 0 Process 21084 Thread 3
---LCB
```

Related Information

The install lwcf handler and sigaction subroutines.

munmap Subroutine

Purpose

Unmaps a mapped region.

Library

Standard C Library (libc.a)

Syntax

```
#include <svs/tvpes.h>
#include <sys/mman.h>
int munmap (addr, len)
void *addr;
size_t len;
```

Description

The munmap subroutine unmaps a mapped file region or anonymous memory region. The munmap subroutine unmaps regions created from calls to the **mmap** subroutine only.

If an address lies in a region that is unmapped by the munmap subroutine and that region is not subsequently mapped again, any reference to that address will result in the delivery of a SIGSEGV signal to the process.

Parameters

addr Specifies the address of the region to be unmapped. Must be a multiple of the page size returned by the sysconf subroutine using the SC PAGE SIZE value for the Name parameter.

Specifies the length, in bytes, of the region to be unmapped. If the len parameter is not a multiple of the len page size returned by the sysconf subroutine using the SC PAGE SIZE value for the Name parameter, the length of the region is rounded up to the next multiple of the page size.

Return Values

When successful, the munmap subroutine returns 0. Otherwise, it returns -1 and sets the errno global variable to indicate the error.

Error Codes

If the **munmap** subroutine is unsuccessful, the **errno** global variable is set to the following value:

EINVAL The addr parameter is not a multiple of the page size as returned by the sysconf subroutine using the SC_PAGE_SIZE value for the Name parameter.

EINVAL The application has requested Single UNIX Specification, Version 2 compliant behavior and the len arguement is 0.

mwakeup Subroutine

Purpose

Wakes up a process that is waiting on a semaphore.

Library

Standard C Library (libc.a)

Syntax

#include <sys/mman.h>
int mwakeup (Sem)
msemaphore * Sem;

Description

The **mwakeup** subroutine wakes up a process that is sleeping and waiting for an idle semaphore. The semaphore should be located in a shared memory region. Use the **mmap** subroutine to create the shared memory section.

All of the values in the **msemaphore** structure must result from a **msem_init** subroutine call. This call may or may not be followed by a sequence of calls to the **msem_lock** subroutine or the **msem_unlock** subroutine. If the **msemaphore** structure value originates in another manner, the results of the **mwakeup** subroutine are undefined.

The address of the **msemaphore** structure is significant. You should be careful not to modify the structure's address. If the structure contains values copied from a **msemaphore** structure at another address, the results of the **mwakeup** subroutine are undefined.

Parameters

Sem Points to the **msemaphore** structure that specifies the semaphore.

Return Values

When successful, the **mwakeup** subroutine returns a value of 0. Otherwise, this routine returns a value of -1 and sets the **errno** global variable to **EFAULT**.

Error Codes

A value of **EFAULT** indicates that the *Sem* parameter points to an invalid address or that the address does not contain a valid **msemaphore** structure.

Related Information

The **mmap** ("mmap or mmap64 Subroutine" on page 690) subroutine, **msem_init** ("msem_init Subroutine" on page 707) subroutine, **msem_lock** ("msem_lock Subroutine" on page 709) subroutine, **msem_unlock** ("msem_unlock Subroutine" on page 711) subroutine, and the **msleep** ("msleep Subroutine" on page 722) subroutine.

Understanding Memory Mapping in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

nan, nanf, or nanl Subroutine

Purpose

Returns quiet NaN.

Syntax

#include <math.h>
double nan (tagp)
const char *tagp;
float nanf (tagp)

```
const char *tagp;
long double nanl (tagp)
const char *tagp;
```

Description

```
The function call nan("n-char-sequence") is equivalent to:
strtod("NAN(n-char-sequence)", (char **) NULL);
The function call nan(" ") is equivalent to:
strtod("NAN()", (char **) NULL)
```

If tagp does not point to an n-char sequence or an empty string, the function call is equivalent to: strtod("NAN", (char **) NULL)

Function calls to nanf and nanl are equivalent to the corresponding function calls to strtof and strtold.

Parameters

Indicates the content of the quiet NaN. tagp

Return Values

The nan, nanf, and nanl subroutines return a quiet NaN with content indicated through tagp.

Related Information

The "atof atoff Subroutine" on page 77.

math.h in AIX 5L Version 5.2 Files Reference.

nearbyint, nearbyintf, or nearbyintl Subroutine

Purpose

Rounds numbers to an integer value in floating-point format.

Syntax

```
#include <math.h>
double nearby int (x)
double x;
float nearby intf (x)
float x;
long double nearby (x)
long double x;
```

Description

The **nearbyint**, **nearbyintf**, and **nearbyintl** subroutines round the x parameter to an integer value in floating-point format, using the current rounding direction and without raising the inexact floating-point exception.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE ALL EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value to be computed. Χ

Return Values

Upon successful completion, the nearbyint, nearbyintf, and nearbyintl subroutines return the rounded integer value.

If x is NaN, a NaN is returned.

If x is ± 0 , ± 0 is returned.

If x is $\pm \ln x$, x is returned.

If the correct value would cause overflow, a range error occurs and the nearbyint, nearbyintf, and nearbyintI subroutines return the value of the macro ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the same sign as x), respectively.

Related Information

"feclearexcept Subroutine" on page 223 and "fetestexcept Subroutine" on page 231.

math.h in AIX 5L Version 5.2 Files Reference.

nextafter, nextafterl, nexttoward, nexttowardf, or nexttowardl **Subroutine**

Purpose

Computes the next representable floating-point number.

Syntax

```
#include <math.h>
float nextafterf (x, y)
float x;
float y;
long double nextafter (x, y)
long double x;
long double y;
double nextafter (x, y)
double x, y;
double nexttoward (x, y)
double x:
long double y;
float nexttowardf (x, y)
float x;
long double y;
```

```
long double nexttoward (x, y)
long double x;
long double y;
```

Description

The **nextafterf**, **nextafterl**, and **nextafter** subroutines compute the next representable floating-point value following x in the direction of y. Thus, if y is less than x, the **nextafter** subroutine returns the largest representable floating-point number less than x.

The **nextafter**, **nextafterf**, and **nextafterl** subroutines return *y* if *x* equals *y*.

The **nexttoward**, **nexttowardf**, and **nexttowardl** subroutines are equivalent to the corresponding nextafter subroutine, except that the second parameter has type long double and the subroutines return y converted to the type of the subroutine if x equals y.

An application wishing to check for error situations should set the errno global variable to zero and call feclearexcept(FE ALL EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE INVALID | FE DIVBYZERO | FE OVERFLOW | FE UNDERFLOW) is nonzero, an error has occurred.

Parameters

- Specifies the starting value. The next representable floating-point number is found from x in the X direction specified by y.
- V Specifies the direction.

Return Values

Upon successful completion, the nextafterf, nextafterf, nextafter, nexttoward, nexttowardf, and **nexttoward** subroutines return the next representable floating-point value following x in the direction of y.

If x==y, y (of the type x) is returned.

If x is finite and the correct function value would overflow, a range error occurs and ±HUGE_VAL, \pm HUGE VALF, and \pm HUGE VALL (with the same sign as x) is returned as appropriate for the return type of the function.

If x or y is NaN, a NaN is returned.

If x!=y and the correct subroutine value is subnormal, zero, or underflows, a range error occurs, and either the correct function value (if representable) or 0.0 is returned.

Error Codes

For the **nextafter** subroutine, if the *x* parameter is finite and the correct function value would overflow, HUGE VAL is returned and errno is set to ERANGE.

Related Information

"feclearexcept Subroutine" on page 223 and "fetestexcept Subroutine" on page 231.

math.h in AIX 5L Version 5.2 Files Reference.

newpass Subroutine

Purpose

Generates a new password for a user.

Library

Security Library (libc.a)

Syntax

#include <usersec.h> #include <userpw.h>

char *newpass(Password) struct userpw *Password;

Description

The **newpass** subroutine generates a new password for the user specified by the *Password* parameter. The new password is then checked to ensure that it meets the password rules on the system unless the user is exempted from these restrictions. Users must have root user authority to invoke this subroutine. The password rules are defined in the /etc/security/user file and are described in both the user file and the **passwd** command.

Passwords can contain almost any legal value for a character but cannot contain (National Language Support (NLS) code points. Passwords cannot have more than the value specified by MAX PASS.

The **newpass** subroutine authenticates the user prior to changing the password. If the **PW ADMCHG** flag is set in the upw flags member of the Password parameter, the supplied password is checked against the password to determine the user corresponding to the real user ID of the process instead of the user specified by the upw name member of the Password parameter structure.

If a password is successfully generated, a pointer to a buffer containing the new password is returned and the last update time is reset.

Note: The newpass subroutine is not safe in a multi-threaded environment. To use newpass in a threaded application, the application must keep the integrity of each thread.

Parameters

Password

Specifies a user password structure. This structure is defined in the userpw.h file and contains the following members:

upw name

A pointer to a character buffer containing the user name.

upw passwd

A pointer to a character buffer containing the current password.

upw_lastupdate

The time the password was last changed, in seconds since the epoch.

upw flags

A bit mask containing 0 or more of the following values:

PW NOCHECK

This bit indicates that new passwords need not meet the composition criteria for passwords on the system.

This bit indicates that password information for this user may only be changed by the root user.

PW ADMCHG

This bit indicates that the password is being changed by an administrator and the password will have to be changed upon the next successful running of the login or su commands to this account.

Security

Policy: Authentication

To change a password, the invoker must be properly authenticated.

Note: Programs that invoke the newpass subroutine should be written to conform to the authentication rules enforced by newpass. The PW_ADMCHG flag should always be explicitly cleared unless the invoker of the command is an administrator.

Return Values

If a new password is successfully generated, a pointer to the new encrypted password is returned. If an error occurs, a null pointer is returned and the errno global variable is set to indicate the error.

Error Codes

The **newpass** subroutine fails if one or more of the following are true;

EINVAL The structure passed to the **newpass** subroutine is invalid.

ESAD Security authentication is denied for the invoker.

EPERM The user is unable to change the password of a user with the PW ADMCHG bit set, and the real user ID

of the process is not the root user.

ENOENT The user is not properly defined in the database.

Related Information

The getpass ("getpass Subroutine" on page 340) subroutine, getuserpw ("getuserpw, putuserpw, or putuserpwhist Subroutine" on page 389) subroutine.

The login command, passwd command, pwdadm command.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

nftw or nftw64 Subroutine

Purpose

Walks a file tree.

Library

Standard C Library (libc.a)

Syntax

```
#include <ftw.h>
int nftw ( Path, Function, Depth, Flags)
const char *Path;
int *(*Function) ( );
int Depth;
int Flags;
int nftw64(Path,Function,Depth)
const char *Path;
int *(*Function) ( );
int Depth;
int Flags;
```

Description

The **nftw** and **nftw64** subroutines recursively descend the directory hierarchy rooted in the *Path* parameter. The nftw and nftw64 subroutines have a similar effect to ftw and ftw64 except that they take an additional argument flags, which is a bitwise inclusive-OR of zero or more of the following flags:

FTW_CHDIR If set, the current working directory will change to each directory as files are reported. If clear, the

current working directory will not change.

FTW_DEPTH If set, all files in a directory will be reported before the directory itself. If clear, the directory will be

reported before any files.

FTW_MOUNT If set, symbolic links will not be followed. If clear the links will be followed.

FTW_PHYS If set, symbolic links will not be followed. If clear the links will be followed, and will not report the

same file more than once.

For each file in the hierarchy, the **nftw** and **nftw64** subroutines call the function specified by the *Function* parameter. The nftw subroutine passes a pointer to a null-terminated character string containing the name of the file, a pointer to a stat structure containing information about the file, an integer and a pointer to an FTW structure. The nftw64 subroutine passes a pointer to a null-terminated character string containing the name of the file, a pointer to a stat64 structure containing information about the file, an integer and a pointer to an FTW structure.

The nftw subroutine uses the stat system call which will fail on files of size larger than 2 Gigabytes. The nftw64 subroutine must be used if there is a possibility of files of size larger than 2 Gigabytes.

The integer passed to the *Function* parameter identifies the file type with one of the following values:

FTW_F Regular file FTW_D Directory

FTW_DNR Directory that cannot be read

FTW_DP The Object is a directory and subdirectories have been visited. (This condition will only occur if

FTW_DEPTH is included in flags).

FTW_SL Symbolic Link

FTW SLN Symbolic Link that does not name an existin file. (This condition will only occur if the FTW PHYS flag

is not included in flags).

FTW_NS File for which the stat structure could not be executed successfully

If the integer is FTW DNR, the files and subdirectories contained in that directory are not processed.

If the integer is FTW NS, the stat structure contents are meaningless. An example of a file that causes FTW NS to be passed to the Function parameter is a file in a directory for which you have read permission but not execute (search) permission.

The FTW structure pointer passed to the Function parameter contains base which is the offset of the object's filename in the pathname passed as the first argument to Function. The value of level indicates depth relative to the root of the walk.

The **nftw** and **nftw64** subroutines use one file descriptor for each level in the tree. The *Depth* parameter specifies the maximum number of file descriptors to be used. In general, the nftw and nftw64 run faster of the value of the Depth parameter is at least as large as the number of levels in the tree. However, the value of the Depth parameter must not be greater than the number of file descriptors currently available for use. If the value of the *Depth* parameter is 0 or a negative number, the effect is the same as if it were 1.

Because the **nftw** and **nftw64** subroutines are recursive, it is possible for it to terminate with a memory fault due to stack overflow when applied to very deep file structures.

The **nftw** and **nftw64** subroutines use the **malloc** subroutine to allocate dynamic storage during its operation. If the **nftw** subroutine is terminated prior to its completion, such as by the **longimp** subroutine being executed by the function specified by the Function parameter or by an interrupt routine, the nftw subroutine cannot free that storage. The storage remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the Function parameter return a nonzero value the next time it is called.

Parameters

Specifies the directory hierarchy to be searched. Path

Function User supplied function that is called for each file encountered.

Depth Specifies the maximum number of file descriptors to be used. Depth cannot be greater than

OPEN_MAX which is described in the sys/limits.h header file.

Return Values

If the tree is exhausted, the **nftw** and **nftw64** subroutine returns a value of 0. If the subroutine pointed to by fn returns a nonzero value, nftw and nftw64 stops its tree traversal and returns whatever value was returned by the subroutine pointed to by fn. If the nftw and nftw64 subroutine detects an error, it returns a -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **nftw** or **nftw64** subroutines detect an error, a value of -1 is returned and the **errno** global variable is set to indicate the error.

The **nftw** and **nftw64** subroutine are unsuccessful if:

EACCES Search permission is denied for any component of the Path parameter or read permission is

denied for Path.

ENAMETOOLONG The length of the path exceeds PATH_MAX while _POSIX_NO_TRUNC is in effect. **ENOENT**

The Path parameter points to the name of a file that does not exist or points to an empty

string.

ENOTDIR A component of the Path parameter is not a directory.

The **nftw** subroutine is unsuccessful if:

EOVERFLOW A file in Path is of a size larger than 2 Gigabytes.

Related Information

The stat or malloc ("malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo heap, alloca, or valloc Subroutine" on page 657) subroutine.

The **ftw** ("ftw or ftw64 Subroutine" on page 278) subroutine.

nl_langinfo Subroutine

Purpose

Returns information on the language or cultural area in a program's locale.

Library

Standard C Library (libc.a)

Syntax

```
#include <nl types.h>
#include <langinfo.h>
char *nl langinfo ( Item)
nl item Item;
```

Description

The nl_langinfo subroutine returns a pointer to a string containing information relevant to the particular language or cultural area defined in the program's locale and corresponding to the *Item* parameter. The active language or cultural area is determined by the default value of the environment variables or by the most recent call to the setlocale subroutine. If the setlocale subroutine has not been called in the program, then the default C locale values will be returned from nl langinfo.

Values for the *Item* parameter are defined in the **langinfo.h** file.

The following table summarizes the categories for which nl_langinfo() returns information, the values the Item parameter can take, and descriptions of the returned strings. In the table, radix character refers to the character that separates whole and fractional numeric or monetary quantities. For example, a period (.) is used as the radix character in the U.S., and a comma (,) is used as the radix character in France.

Category	Value of item	Returned Result
LC_MONETARY	CRNCYSTR	Currency symbol and its position.
LC_NUMERIC	RADIXCHAR	Radix character.

Category	Value of item	Returned Result
LC_NUMERIC	THOUSEP	Separator for the thousands.
LC_MESSAGES	YESSTR	Affirmative response for yes/no queries.
LC_MESSAGES	NOSTR	Negative response for yes/no queries.
LC_TIME	D_T_FMT	String for formatting date and time.
LC_TIME	D_FMT	String for formatting date.
LC_TIME	T_FMT	String for formatting time.
LC_TIME	AM_STR	Antemeridian affix.
LC_TIME	PM_STR	Postmeridian affix.
LC_TIME	DAY_1 through DAY_7	Name of the first day of the week to the seventh day of the week.
LC_TIME	ABDAY_1 through ABDAY-7	Abbreviated name of the first day of the week to the seventh day of the week.
LC_TIME	MON_1 through MON_12	Name of the first month of the year to the twelfth month of the year.
LC_TIME	ABMON_1 through ABMON_12	Abbreviated name of the first month of the year to the twelfth month.
LC_CTYPE	CODESET	Code set currently in use in the program.

Note: The information returned by the nl_langinfo subroutine is located in a static buffer. The contents of this buffer are overwritten in subsequent calls to the nl langinfo subroutine. Therefore, you should save the returned information.

Parameter

Item Information needed from locale.

Return Values

In a locale where language information data is not defined, the nl_langinfo subroutine returns a pointer to the corresponding string in the C locale. In all locales, the nl_langinfo subroutine returns a pointer to an empty string if the *Item* parameter contains an invalid setting.

The nl_langinfo subroutine returns a pointer to a static area. Subsequent calls to the nl_langinfo subroutine overwrite the results of a previous call.

Related Information

The localeconv ("localeconv Subroutine" on page 625) subroutine, rpmatch subroutine, setlocale subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview and Setting the Locale in AIX 5L Version 5.2 National Language Support Guide and Reference.

nlist, nlist64 Subroutine

Purpose

Gets entries from a name list.

Library

Standard C Library (libc.a)

Berkeley Compatibility Library [libbsd.a] for the nlist subroutine, 32-bit programs, and POWER-based platforms

Syntax

```
#include <nlist.h>
int nlist ( FileName, NL )
const char *FileName;
struct nlist *NL;
int nlist64 ( FileName, NL64 )
const char *FileName;
struct nlist64 *NL64;
```

Description

The **nlist** and **nlist64** subroutines examine the name list in the object file named by the *FileName* parameter. The subroutine selectively reads a list of values and stores them into an array of **nlist** or **nlist64** structures pointed to by the *NL* or *NL64* parameter, respectively.

The name list specified by the *NL* or *NL64* parameter consists of an array of **nlist** or **nlist64** structures containing symbol names and other information. The list is terminated with an element that has a null pointer or a pointer to a null string in the **n_name** structure member. Each symbol name is looked up in the name list of the file. If the name is found, the value of the symbol is stored in the structure, and the other fields are filled in. If the program was not compiled with the **-g** flag, the **n_type** field may be 0.

If multiple instances of a symbol are found, the information about the last instance is stored. If a symbol is not found, all structure fields except the **n name** field are set to 0. Only global symbols will be found.

The **nlist** and **nlist64** subroutines run in both 32-bit and 64-bit programs that read the name list of both 32-bit and 64-bit object files, with one exception: in 32-bit programs, **nlist** will return -1 if the specified file is a 64-bit object file.

The **nlist** and **nlist64** subroutines are used to read the name list from XCOFF object files.

The **nlist64** subroutine can be used to examine the system name list kept in the kernel, by specifying **/unix** as the *FileName* parameter. The **knlist** subroutine can also be used to look up symbols in the current kernel namespace.

Note: The nlist.h header file has a #define field for n_name. If a source file includes both nlist.h and netdb.h, there will be a conflict with the use of n_name. If netdb.h is included after nlist.h, n_name will be undefined. To correct this problem, _n._n_name should be used instead. If netdb.h is included before nlist.h, and you need to refer to the n_name field of struct netent, you should undefine n_name by entering:

```
#undef n name
```

The **nlist** subroutine in **libbsd.a** is supported only in 32-bit mode.

Parameters

FileName Specifies the name of the file containing a name list.

NI Points to the array of nlist structures. NI 64 Points to the array of nlist64 structures.

Return Values

Upon successful completion, a 0 is returned, even if some symbols could not be found. In the libbsd.a version of **nlist**, the number of symbols not found in the object file's name list is returned. If the file cannot be found or if it is not a valid name list, a value of -1 is returned.

Compatibility Interfaces

To obtain the BSD-compatible version of the subroutine 32-bit applications, compile with the libbsd.a library by using the -lbsd flag.

Related Information

The knlist subroutine.

The a.out file in XCOFF format.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

ns addr Subroutine

Purpose

XNS address conversion routines.

Library

Standard C Library (libc.a)

Syntax

#include <sys/types.h> #include <netns/ns.h>

struct ns_addr(char *cp)

Description

The ns_addr subroutine interprets character strings representing XNS addresses, returning binary information suitable for use in system calls.

The ns_addr subroutine separates an address into one to three fields using a single delimiter and examines each field for byte separators (colon or period). The delimiters are:

- period
- colon
- pound sign.

If byte separators are found, each subfield separated is taken to be a small hexadecimal number, and the entirety is taken as a network-byte-ordered quantity to be zero extended in the high-networked-order bytes. Next, the field is inspected for hyphens, which would indicate the field is a number in decimal notation with hyphens separating the millenia. The field is assumed to be a number, interpreted as hexadecimal, if a leading Ox (as in C), a trailing H, (as in Mesa), or any super-octal digits are present. The field is interpreted as octal if a leading Ox is present and there are no super-octal digits. Otherwise, the field is converted as a decimal number.

Parameter

cp Returns a pointer to the address of a **ns_addr** structure.

ns_ntoa Subroutine

Purpose

XNS address conversion routines.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/types.h>
#include <netns/ns.h>
char *ns_ntoa (
struct ns addr ns)
```

Description

The **ns_ntoa** subroutine takes XNS addresses and returns ASCII strings representing the address in a notation in common use in the Xerox Development Environment:

```
<network number> <host number> <port number>
```

Trailing zero fields are suppressed, and each number is printed in hexadecimal, in a format suitable for input to the **ns addr** subroutine. Any fields lacking super-decimal digits will have a trailing *H* appended.

Note: The string returned by **ns_ntoa** resides in static memory.

Parameter

ns Returns a pointer to a string.

odm_add_obj Subroutine

Purpose

Adds a new object into an object class.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

int odm add obj (ClassSymbol, DataStructure) CLASS SYMBOL ClassSymbol; struct ClassName *DataStructure;

Description

The odm_add_obj subroutine takes as input the class symbol that identifies both the object class to add and a pointer to the data structure containing the object to be added.

The odm_add_obj subroutine opens and closes the object class around the subroutine if the object class was not previously opened. If the object class was previously opened, the subroutine leaves the object class open when it returns.

Parameters

ClassSymbol Specifies a class symbol identifier returned from an odm open class

> subroutine. If the odm_open_class subroutine has not been called, then this identifier is the ClassName_CLASS structure that was created by the

odmcreate command.

DataStructure Specifies a pointer to an instance of the C language structure corresponding to

the object class referenced by the ClassSymbol parameter. The structure is declared in the .h file created by the odmcreate command and has the same

name as the object class.

Return Values

Upon successful completion, an identifier for the object that was added is returned. If the odm_add_obj subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm add obj** subroutine sets the **odmerrno** variable to one of the following error codes:

- ODMI_CLASS_DNE
- ODMI CLASS PERMS
- ODMI INVALID CLXN
- ODMI INVALID PATH
- ODMI MAGICNO ERR
- ODMI OPEN ERR
- ODMI PARAMS
- ODMI_READ_ONLY
- ODMI TOOMANYCLASSES

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The odm_create_class ("odm_create_class Subroutine" on page 744) subroutine, odm_open_class ("odm_open_class Subroutine" on page 755) subroutine, odm_rm_obj ("odm_rm_obj Subroutine" on page 758) subroutine.

The odmcreate command.

See ODM Example Code and Output in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_change_obj Subroutine

Purpose

Changes an object in the object class.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

int odm_change_obj (ClassSymbol, DataStructure) **CLASS SYMBOL** ClassSymbol; struct ClassName *DataStructure;

Description

The **odm change obj** subroutine takes as input the class symbol that identifies both the object class to change and a pointer to the data structure containing the object to be changed. The application program must first retrieve the object with an **odm_get_obj** subroutine call, change the data values in the returned structure, and then pass that structure to the **odm_change_obj** subroutine.

The odm_change_obj subroutine opens and closes the object class around the change if the object class was not previously opened. If the object class was previously opened, then the subroutine leaves the object class open when it returns.

Parameters

Specifies a class symbol identifier returned from an odm open class subroutine. If the ClassSymbol

> odm_open_class subroutine has not been called, then this identifier is the ClassName_CLASS structure that is created by the odmcreate command.

DataStructure Specifies a pointer to an instance of the C language structure corresponding to the object

class referenced by the ClassSymbol parameter. The structure is declared in the .h file created by the **odmcreate** command and has the same name as the object class.

Return Values

Upon successful completion, a value of 0 is returned. If the odm_change_obj subroutine fails, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the odm_change_obj subroutine sets the odmerrno variable to one of the following error codes:

- ODMI CLASS DNE
- ODMI CLASS PERMS
- ODMI INVALID CLXN
- ODMI INVALID PATH

- ODMI MAGICNO ERR
- ODMI NO OBJECT
- ODMI_OPEN_ERR
- ODMI_PARAMS
- ODMI READ ONLY
- ODMI_TOOMANYCLASSES

Related Information

The **odm_get_obj** ("odm_get_obj, odm_get_first, or odm_get_next Subroutine" on page 750) subroutine.

The **odmchange** command, **odmcreate** command.

See ODM Example Code and Output in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_close_class Subroutine

Purpose

Closes an ODM object class.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

int odm_close_class (ClassSymbol) CLASS SYMBOL ClassSymbol;

Description

The **odm close class** subroutine closes the specified object class.

Parameters

ClassSymbol

Specifies a class symbol identifier returned from an odm_open_class subroutine. If the odm_open_class subroutine has not been called, then this identifier is the ClassName_CLASS structure that was created by the odmcreate command.

Return Values

Upon successful completion, a value of 0 is returned. If the odm_close_class subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the odm_close_class subroutine sets the odmerrno variable to one of the following error codes:

- ODMI_CLASS_DNE
- ODMI CLASS PERMS
- ODMI_INVALID_CLXN
- ODMI_INVALID_PATH
- ODMI_MAGICNO_ERR
- ODMI_OPEN_ERR
- ODMI_TOOMANYCLASSES

Related Information

The odm_open_class ("odm_open_class Subroutine" on page 755) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_create_class Subroutine

Purpose

Creates an object class.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

int odm_create_class (ClassSymbol)
CLASS_SYMBOL ClassSymbol;

Description

The **odm_create_class** subroutine creates an object class. However, the **.c** and **.h** files generated by the **odmcreate** command are required to be part of the application.

Parameters

ClassSymbol

Specifies a class symbol of the form *ClassName_CLASS*, which is declared in the .h file created by the **odmcreate** command.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_create_class** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_create_class** subroutine sets the **odmerrno** variable to one of the following error codes:

- ODMI CLASS EXISTS
- ODMI CLASS PERMS
- ODMI_INVALID_CLXN

- ODMI INVALID PATH
- ODMI MAGICNO ERR
- ODMI_OPEN_ERR

Related Information

The **odm_mount_class** ("odm_mount_class Subroutine" on page 753) subroutine.

The **odmcreate** command.

See ODM Example Code and Output in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_err_msg Subroutine

Purpose

Returns an error message string.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

int odm err msg (ODMErrno, MessageString) long ODMErrno; char **MessageString;

Description

The **odm err msq** subroutine takes as input an *ODMErrno* parameter and an address in which to put the string pointer of the message string that corresponds to the input ODM error number. If no corresponding message is found for the input error number, a null string is returned and the subroutine is unsuccessful.

Parameters

ODMErrno Specifies the error code for which the message string is retrieved.

MessageString Specifies the address of a string pointer that will point to the returned error message string.

Return Values

Upon successful completion, a value of 0 is returned. If the odm_err_msg subroutine is unsuccessful, a value of -1 is returned, and the MessageString value returned is a null string.

Examples

The following example shows the use of the **odm_err_msg** subroutine:

```
#include <odmi.h>
char *error_message;
```

Related Information

Appendix B, "ODM Error Codes," on page 1003 in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1 describes error codes.

See Appendix B, Appendix B, "ODM Error Codes," on page 1003 for explanations of the ODM error codes.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_free_list Subroutine

Purpose

Frees memory previously allocated for an odm_get_list subroutine.

Library

Object Data Manager Library (libodm.a)

Syntax

```
#include <odmi.h>
int odm_free_list ( ReturnData, DataInfo)
struct ClassName *ReturnData;
struct listinfo *DataInfo;
```

Description

The **odm_free_list** subroutine recursively frees up a tree of memory object lists that were allocated for an **odm_get_list** subroutine.

Parameters

ReturnData DataInfo Points to the array of *ClassName* structures returned from the **odm_get_list** subroutine. Points to the **listinfo** structure that was returned from the **odm_get_list** subroutine. The **listinfo** structure has the following form:

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_free_list** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the odm_free_list subroutine sets the odmerrno variable to one of the following error codes:

- ODMI MAGICNO ERR
- ODMI PARAMS

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The odm_get_list ("odm_get_list Subroutine" on page 748) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_get_by_id Subroutine

Purpose

Retrieves an object from an ODM object class by its ID.

Library

Object Data Manager Library (libodm.a)

Syntax

```
#include <odmi.h>
```

```
struct ClassName *odm get by id( ClassSymbol, ObjectID, ReturnData)
CLASS SYMBOL ClassSymbol;
int ObjectID;
struct ClassName *ReturnData;
```

Description

The **odm** get by id subroutine retrieves an object from an object class. The object to be retrieved is specified by passing its ObjectID parameter from its corresponding ClassName structure.

Parameters

ClassSymbol Specifies a class symbol identifier of the form ClassName_CLASS, which is declared in the .h

file created by the odmcreate command.

ObjectID Specifies an identifier retrieved from the corresponding ClassName structure of the object

ReturnData Specifies a pointer to an instance of the C language structure corresponding to the object class

referenced by the ClassSymbol parameter. The structure is declared in the .h file created by

the odmcreate command and has the same name as the object class.

Return Values

Upon successful completion, a pointer to the ClassName structure containing the object is returned. If the odm get by id subroutine is unsuccessful, a value of -1 is returned and the odmerrno variable is set to an error code.

Error Codes

Failure of the odm_get_by_id subroutine sets the odmerrno variable to one of the following error codes:

- ODMI CLASS DNE
- ODMI_CLASS_PERMS
- ODMI_INVALID_CLXN
- ODMI_INVALID_PATH
- ODMI_MAGICNO_ERR
- ODMI_MALLOC_ERR
- ODMI_NO_OBJECT
- ODMI_OPEN_ERR
- ODMI PARAMS
- ODMI_TOOMANYCLASSES

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The odm_get_obj ("odm_get_obj, odm_get_first, or odm_get_next Subroutine" on page 750), odm get first ("odm get obj, odm get first, or odm get next Subroutine" on page 750), or odm get next ("odm get obj, odm get first, or odm get next Subroutine" on page 750) subroutine.

The **odmcreate** command.

See ODM Example Code and Output in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_get_list Subroutine

Purpose

Retrieves all objects in an object class that match the specified criteria.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

```
struct ClassName *odm_get_list (ClassSymbol, Criteria, ListInfo, MaxReturn, LinkDepth)
struct ClassName_CLASS ClassSymbol;
char * Criteria;
struct listinfo * ListInfo;
int MaxReturn, LinkDepth;
```

Description

The odm_get_list subroutine takes an object class and criteria as input, and returns a list of objects that satisfy the input criteria. The subroutine opens and closes the object class around the subroutine if the object class was not previously opened. If the object class was previously opened, the subroutine leaves the object class open when it returns.

Parameters

ClassSymbol Specifies a class symbol identifier returned from an odm_open_class subroutine. If the

odm_open_class subroutine has not been called, then this is the ClassName_CLASS

structure created by the odmcreate command.

Criteria Specifies a string that contains the qualifying criteria for selecting the objects to remove. ListInfo

Specifies a structure containing information about the retrieval of the objects. The listinfo

structure has the following form:

struct listinfo { char ClassName[16]; /* class name used for query */ char criteria[256]; /* query criteria */ /* number of matches found */ int num; /* for ODM use */ int valid;

/* symbol for queried class */ CLASS SYMBOL class;

MaxReturn Specifies the expected number of objects to be returned. This is used to control the increments

in which storage for structures is allocated, to reduce the realloc subroutine copy overhead.

Specifies the number of levels to recurse for objects with ODM_LINK descriptors. A setting of 1 LinkDepth

indicates only the top level is retrieved; 2 indicates ODM_LINKs will be followed from the top/first level only: 3 indicates ODM_LINKs will be followed at the first and second level, and so

on.

Return Values

Upon successful completion, a pointer to an array of C language structures containing the objects is returned. This structure matches that described in the .h file that is returned from the odmcreate command. If no match is found, null is returned. If the odm get list subroutine fails, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the odm_get_list subroutine sets the odmerrno variable to one of the following error codes:

- ODMI BAD CRIT
- ODMI CLASS DNE
- ODMI_CLASS_PERMS
- ODMI_INTERNAL_ERR
- ODMI_INVALID_CLXN
- ODMI_INVALID_PATH
- ODMI_LINK_NOT_FOUND
- ODMI_MAGICNO_ERR
- ODMI MALLOC ERR
- ODMI_OPEN_ERR
- ODMI PARAMS
- ODMI TOOMANYCLASSES

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The **odm get by id** ("odm get by id Subroutine" on page 747) subroutine, **odm get obj** ("odm get obj, odm get first, or odm get next Subroutine" on page 750) subroutine, odm open class ("odm_open_class Subroutine" on page 755) subroutine, or odm_free_list ("odm_free_list Subroutine" on page 746) subroutine.

The **odmcreate** command, **odmget** command.

For information on qualifying criteria, see "Understanding ODM Object Searches" in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

See ODM Example Code and Output in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm get obj, odm get first, or odm get next Subroutine

Purpose

Retrieves objects, one object at a time, from an ODM object class.

Library

Object Data Manager Library (libodm.a)

Syntax

```
#include <odmi.h>
struct ClassName *odm get obj ( ClassSymbol, Criteria, ReturnData, FIRST NEXT)
struct ClassName *odm get first (ClassSymbol, Criteria, ReturnData)
struct ClassName *odm_get_next (ClassSymbol, ReturnData)
CLASS SYMBOL ClassSymbol;
char *Criteria;
struct ClassName *ReturnData;
int FIRST NEXT;
```

Description

The odm get obj, odm get first, and odm get next subroutines retrieve objects from ODM object classes and return the objects into C language structures defined by the .h file produced by the odmcreate command.

The odm get obj, odm get first, and odm get next subroutines open and close the specified object class if the object class was not previously opened. If the object class was previously opened, the subroutines leave the object class open upon return.

Parameters

Specifies a class symbol identifier returned from an odm_open_class subroutine. If the ClassSymbol

> odm_open_class subroutine has not been called, then this identifier is the ClassName_CLASS structure that was created by the odmcreate command.

Criteria Specifies the string that contains the qualifying criteria for retrieval of the objects. ReturnData

Specifies the pointer to the data structure in the .h file created by the odmcreate command. The name of the structure in the .h file is ClassName. If the ReturnData parameter is null (ReturnData == null), space is allocated for the parameter and the calling application is responsible for freeing this space at a later time.

If variable length character strings (vchar) are returned, they are referenced by pointers in the ReturnData structure. Calling applications must free each vchar between each call to the **odm get** subroutines; otherwise storage will be lost.

FIRST_NEXT

Specifies whether to get the first object that matches the criteria or the next object. Valid values are:

ODM_FIRST

Retrieve the first object that matches the search criteria.

ODM_NEXT

Retrieve the next object that matches the search criteria. The Criteria parameter is ignored if the FIRST_NEXT parameter is set to **ODM_NEXT**.

Return Values

Upon successful completion, a pointer to the retrieved object is returned. If no match is found, null is returned. If an odm_get_obj, odm_get_first, or odm_get_next subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the odm_get_obj, odm_get_first or odm_get_next subroutine sets the odmerrno variable to one of the following error codes:

- ODMI BAD CRIT
- ODMI CLASS DNE
- ODMI_CLASS_PERMS
- ODMI_INTERNAL_ERR
- ODMI_INVALID_CLXN
- ODMI INVALID PATH
- ODMI MAGICNO ERR
- ODMI MALLOC ERR
- ODMI OPEN ERR
- ODMI_TOOMANYCLASSES

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The odm_get_list ("odm_get_list Subroutine" on page 748) subroutine, odm_open_class ("odm open class Subroutine" on page 755) subroutine, odm rm by id ("odm rm by id Subroutine" on page 756) subroutine, **odm rm obj** ("odm rm obj Subroutine" on page 758) subroutine.

The **odmcreate** command, **odmget** command.

For more information about qualifying criteria, see "Understanding ODM Object Searches" in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

See ODM Example Code and Output in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm initialize Subroutine

Purpose

Prepares ODM for use by an application.

Library

Object Data Manager Library (libodm.a)

Syntax

```
#include <odmi.h>
int odm_initialize( )
```

Description

The **odm initialize** subroutine starts ODM for use with an application program.

Return Values

Upon successful completion, a value of 0 is returned. If the odm initialize subroutine is unsuccessful, a value of -1 is returned and the odmerrno variable is set to an error code.

Error Codes

Failure of the odm_initialize subroutine sets the odmerrno variable to one of the following error codes:

- ODMI INVALID PATH
- ODMI_MALLOC_ERR

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The odm_terminate ("odm_terminate Subroutine" on page 762) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm lock Subroutine

Purpose

Puts an exclusive lock on the requested path name.

Library

Object Data Manager Library (libodm.a)

Syntax

```
#include <odmi.h>
int odm_lock ( LockPath, TimeOut)
char *LockPath;
int TimeOut:
```

Description

The odm_lock subroutine is used by an application to prevent other applications or methods from accessing an object class or group of object classes. A lock on a directory path name does not prevent another application from acquiring a lock on a subdirectory or object class within that directory.

Note: Coordination of locking is the responsibility of the application accessing the object classes.

The **odm** lock subroutine returns a lock identifier that is used to call the **odm** unlock subroutine.

Parameters

LockPath Specifies a string containing the path name in the file system in which to locate object classes or the

path name to an object class to lock.

TimeOut Specifies the amount of time, in seconds, to wait if another application or method holds a lock on the requested object class or classes. The possible values for the *TimeOut* parameter are:

TimeOut = **ODM_NOWAIT**

The **odm lock** subroutine is unsuccessful if the lock cannot be granted immediately.

TimeOut = Integer

The odm_lock subroutine waits the specified amount of seconds to retry an unsuccessful lock request.

TimeOut = **ODM WAIT**

The odm lock subroutine waits until the locked path name is freed from its current lock and then locks it.

Return Values

Upon successful completion, a lock identifier is returned. If the odm_lock subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the odm_lock subroutine sets the odmerrno variable to one of the following error codes:

- ODMI_BAD_LOCK
- ODMI BAD TIMEOUT
- ODMI BAD TOKEN
- ODMI LOCK BLOCKED
- ODMI LOCK ENV
- ODMI_MALLOC_ERR
- ODMI_UNLOCK

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The **odm_unlock** ("odm_unlock Subroutine" on page 763) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_mount_class Subroutine

Purpose

Retrieves the class symbol structure for the specified object class name.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

CLASS_SYMBOL odm_mount_class (ClassName)
char *ClassName;

Description

The **odm_mount_class** subroutine retrieves the class symbol structure for a specified object class. The subroutine can be called by applications (for example, the ODM commands) that have no previous knowledge of the structure of an object class before trying to access that class. The **odm_mount_class** subroutine determines the class description from the object class header information and creates a **CLASS_SYMBOL** object class that is returned to the caller.

The object class is not opened by the **odm_mount_class** subroutine. Calling the subroutine subsequent times for an object class that is already open or mounted returns the same **CLASS_SYMBOL** object class.

Mounting a class that links to another object class recursively mounts to the linked class. However, if the recursive mount is unsuccessful, the original **odm_mount_class** subroutine does not fail; the **CLASS_SYMBOL** object class is set up with a null link.

Parameters

ClassName Specifies the name of an object class from which to retrieve the class description.

Return Values

Upon successful completion, a **CLASS_SYMBOL** is returned. If the **odm_mount_class** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_mount_class** subroutine sets the **odmerrno** variable to one of the following error codes:

- ODMI BAD CLASSNAME
- ODMI_BAD_CLXNNAME
- ODMI_CLASS_DNE
- ODMI CLASS PERMS
- ODMI_CLXNMAGICNO_ERR
- ODMI INVALID CLASS
- ODMI INVALID CLXN
- ODMI_MAGICNO_ERR
- ODMI_MALLOC_ERR
- ODMI_OPEN_ERR
- ODMI PARAMS
- ODMI TOOMANYCLASSES
- ODMI TOOMANYCLASSES

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The odm_create_class ("odm_create_class Subroutine" on page 744) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_open_class Subroutine

Purpose

Opens an ODM object class.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

```
CLASS SYMBOL odm open class ( ClassSymbol)
CLASS SYMBOL ClassSymbol;
```

Description

The odm_open_class subroutine can be called to open an object class. Most subroutines implicitly open a class if the class is not already open. However, an application may find it useful to perform an explicit open if, for example, several operations must be done on one object class before closing the class.

Parameter

ClassSymbol

Specifies a class symbol of the form ClassName CLASS that is declared in the .h file created by the odmcreate command.

Return Values

Upon successful completion, a ClassSymbol parameter for the object class is returned. If the odm_open_class subroutine is unsuccessful, a value of -1 is returned and the odmerrno variable is set to an error code.

Error Codes

Failure of the odm_open_class subroutine sets the odmerrno variable to one of the following error codes:

- ODMI CLASS DNE
- ODMI CLASS PERMS
- ODMI_INVALID_PATH
- ODMI MAGICNO ERR
- ODMI_OPEN_ERR
- ODMI TOOMANYCLASSES

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The odm close class ("odm close class Subroutine" on page 743) subroutine.

The **odmcreate** command.

See ODM Example Code and Output in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs for an example of a .h file.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_rm_by_id Subroutine

Purpose

Removes objects specified by their IDs from an ODM object class.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

int odm_rm_by_id(ClassSymbol, ObjectID)
CLASS_SYMBOL ClassSymbol;
int ObjectID;

Description

The **odm_rm_by_id** subroutine is called to delete an object from an object class. The object to be deleted is specified by passing its object ID from its corresponding *ClassName* structure.

Parameters

ClassSymbol Identifies a class symbol returned from an odm_open_class subroutine. If the

odm_open_class subroutine has not been called, this is the ClassName_CLASS structure that

was created by the odmcreate command.

ObjectID Identifies the object. This information is retrieved from the corresponding ClassName structure

of the object class.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_rm_by_id** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_rm_by_id** subroutine sets the **odmerrno** variable to one of the following error codes:

- ODMI_CLASS_DNE
- ODMI_CLASS_PERMS
- ODMI_FORK
- ODMI_INVALID_CLXN
- ODMI_INVALID_PATH
- ODMI_MAGICNO_ERR
- ODMI_MALLOC_ERR
- ODMI_NO_OBJECT
- ODMI_OPEN_ERR
- ODMI_OPEN_PIPE
- ODMI_PARAMS
- ODMI_READ_ONLY

- ODMI READ PIPE
- ODMI TOOMANYCLASSES
- ODMI_TOOMANYCLASSES

Related Information

The odm_get_obj ("odm_get_obj, odm_get_first, or odm_get_next Subroutine" on page 750) subroutine, odm_open_class ("odm_open_class Subroutine" on page 755) subroutine.

The **odmdelete** command.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm rm class Subroutine

Purpose

Removes an object class from the file system.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

int odm rm class (ClassSymbol) CLASS SYMBOL ClassSymbol;

Description

The odm_rm_class subroutine removes an object class from the file system. All objects in the specified class are deleted.

Parameter

ClassSymbol

Identifies a class symbol returned from the odm_open_class subroutine. If the odm_open_class subroutine has not been called, this is the ClassName_CLASS structure created by the odmcreate command.

Return Values

Upon successful completion, a value of 0 is returned. If the odm rm class subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_rm_class** subroutine sets the **odmerrno** variable to one of the following error codes:

- ODMI_CLASS_DNE
- ODMI_CLASS_PERMS
- ODMI_INVALID_CLXN
- ODMI_INVALID_PATH

- ODMI_MAGICNO_ERR
- ODMI OPEN ERR
- ODMI_TOOMANYCLASSES
- ODMI_UNLINKCLASS_ERR
- ODMI_UNLINKCLXN_ERR

Related Information

The odm_open_class ("odm_open_class Subroutine" on page 755) subroutine.

The odmcreate command, odmdrop command.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_rm_obj Subroutine

Purpose

Removes objects from an ODM object class.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

int odm_rm_obj (ClassSymbol, Criteria)
CLASS_SYMBOL ClassSymbol;
char *Criteria;

Description

The **odm rm obj** subroutine deletes objects from an object class.

Parameters

ClassSymbol Identifies a class symbol returned from an odm_open_class subroutine. If the

odm_open_class subroutine has not been called, this is the ClassName_CLASS structure that

was created by the odmcreate command.

Criteria Contains as a string the qualifying criteria for selecting the objects to remove.

Return Values

Upon successful completion, the number of objects deleted is returned. If the **odm_rm_obj** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_rm_obj** subroutine sets the **odmerrno** variable to one of the following error codes:

- ODMI_BAD_CRIT
- ODMI_CLASS_DNE

- ODMI CLASS PERMS
- ODMI FORK
- ODMI_INTERNAL_ERR
- ODMI_INVALID_CLXN
- ODMI INVALID PATH
- ODMI_MAGICNO_ERR
- ODMI_MALLOC_ERR
- ODMI OPEN ERR
- ODMI OPEN PIPE
- ODMI_PARAMS
- ODMI_READ_ONLY
- ODMI_READ_PIPE
- ODMI_TOOMANYCLASSES

Related Information

The odm add obj ("odm add obj Subroutine" on page 740) subroutine, odm open class ("odm open class Subroutine" on page 755) subroutine.

The **odmcreate** command, **odmdelete** command.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

For information on qualifying criteria, see "Understanding ODM Object Searches" in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm run method Subroutine

Purpose

Runs a specified method.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

```
int odm run method (MethodName, MethodParameters, NewStdOut, NewStdError)
char * MethodName, * MethodParameters;
char ** NewStdOut, ** NewStdError;
```

Description

The odm_run_method subroutine takes as input the name of the method to run, any parameters for the method, and addresses of locations for the odm_run_method subroutine to store pointers to the stdout (standard output) and stderr (standard error output) buffers. The application uses the pointers to access the stdout and stderr information generated by the method.

Parameters

MethodName Indicates the method to execute. The method can already be known by the

applications, or can be retrieved as part of an odm_get_obj subroutine call.

MethodParameters Specifies a list of parameters for the specified method.

NewStdOut Specifies the address of a pointer to the memory where the standard output of the

method is stored. If the NewStdOut parameter points to a null value (*NewStdOut ==

NULL), standard output is not captured.

NewStdFrror Specifies the address of a pointer to the memory where the standard error output of

the method will be stored. If the NewStdError parameter points to a null value

(*NewStdError == NULL), standard error output is not captured.

Return Values

If successful, the odm_run_method subroutine returns the exit status and out_ptr and err_ptr should contain the relevant information. If unsuccessful, the odm run method subroutine will return -1 and set the **odmerrno** variable to an error code.

Note: AIX methods usually return the exit code defined in the cf.h file if the methods exit on error.

Error Codes

Failure of the odm run method subroutine sets the odmerrno variable to one of the following error codes:

- ODMI FORK
- ODMI_MALLOC_ERR
- ODMI_OPEN_PIPE
- ODMI PARAMS
- ODMI_READ_PIPE

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The **odm_get_obj** ("odm_get_obj, odm_get_first, or odm_get_next Subroutine" on page 750) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_set_path Subroutine

Purpose

Sets the default path for locating object classes.

Library

Object Data Manager Library (libodm.a)

Syntax

```
#include <odmi.h>
```

```
char *odm set path ( NewPath)
char *NewPath;
```

Description

The odm_set_path subroutine is used to set the default path for locating object classes. The subroutine allocates memory, sets the default path, and returns the pointer to memory. Once the operation is complete, the calling application should free the pointer using the free ("malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, or valloc Subroutine" on page 657) subroutine.

Parameters

NewPath Contains, as a string, the path name in the file system in which to locate object classes.

Return Values

Upon successful completion, a string pointing to the previous default path is returned. If the odm set path subroutine is unsuccessful, a value of -1 is returned and the odmerrno variable is set to an error code.

Error Codes

Failure of the **odm set path** subroutine sets the **odmerrno** variable to one of the following error codes:

- ODMI INVALID PATH
- ODMI_MALLOC_ERR

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The free ("malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, or valloc Subroutine" on page 657) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm set perms Subroutine

Purpose

Sets the default permissions for an ODM object class at creation time.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

```
int odm set perms ( NewPermissions)
int NewPermissions;
```

Description

The **odm_set_perms** subroutine defines the default permissions to assign to object classes at creation.

Parameters

NewPermissions

Specifies the new default permissions parameter as an integer.

Return Values

Upon successful completion, the current default permissions are returned. If the odm_set_perms subroutine is unsuccessful, a value of -1 is returned.

Related Information

See Appendix B, "ODM Error Codes," on page 1003 for explanations of the ODM error codes.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm terminate Subroutine

Purpose

Terminates an ODM session.

Library

Object Data Manager Library (libodm.a)

Syntax

```
#include <odmi.h>
int odm terminate ( )
```

Description

The odm terminate subroutine performs the cleanup necessary to terminate an ODM session. After running an odm terminate subroutine, an application must issue an odm initialize subroutine to resume ODM operations.

Return Values

Upon successful completion, a value of 0 is returned. If the odm terminate subroutine is unsuccessful, a value of -1 is returned and the odmerrno variable is set to an error code.

Error Codes

Failure of the odm_terminate subroutine sets the odmerrno variable to one of the following error codes:

- ODMI CLASS DNE
- ODMI_CLASS_PERMS
- ODMI_INVALID_CLXN
- ODMI_INVALID_PATH
- ODMI_LOCK_ID
- ODMI_MAGICNO_ERR
- ODMI_OPEN_ERR
- ODMI_TOOMANYCLASSES
- ODMI_UNLOCK

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The **odm** initialize ("odm initialize Subroutine" on page 751) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

odm_unlock Subroutine

Purpose

Releases a lock put on a path name.

Library

Object Data Manager Library (libodm.a)

Syntax

#include <odmi.h>

int odm_unlock (LockID) int LockID:

Description

The odm_unlock subroutine releases a previously granted lock on a path name. This path name can be a directory containing subdirectories and object classes.

Parameters

LockID Identifies the lock returned from the **odm_lock** subroutine.

Return Values

Upon successful completion a value of 0 is returned. If the odm_unlock subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm unlock** subroutine sets the **odmerrno** variable to one of the following error codes:

- ODMI LOCK ID
- ODMI UNLOCK

See Appendix B, "ODM Error Codes" for explanations of the ODM error codes.

Related Information

The odm_lock ("odm_lock Subroutine" on page 752) subroutine.

Object Data Manager (ODM) Overview for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

open, openx, open64, creat, or creat64 Subroutine

Purpose

Opens a file for reading or writing.

Syntax

#include <fcntl.h>

```
int open ( Path, OFlag [, Mode])
const char *Path;
int OFlag;
mode t Mode;
int openx (Path, OFlag, Mode, Extension)
const char *Path;
int OFlag;
mode t Mode;
int Extension;
int creat (Path, Mode)
const char *Path;
mode_t Mode;
Note: The open64 and creat64 subroutines apply to AIX 4.2 and later releases.
int open64 (Path, OFlag [, Mode])
const char *Path;
int OFlag;
mode t Mode;
int creat64 (Path, Mode)
const char *Path;
mode t Mode;
```

Description

Note: The **open64** and **creat64** subroutines apply to AIX 4.2 and later releases.

The open, openx, and creat subroutines establish a connection between the file named by the Path parameter and a file descriptor. The opened file descriptor is used by subsequent I/O subroutines, such as read and write, to access that file.

The **openx** subroutine is the same as the **open** subroutine, with the addition of an *Extension* parameter, which is provided for device driver use. The **creat** subroutine is equivalent to the **open** subroutine with the O_WRONLY, O_CREAT, and O_TRUNC flags set.

The returned file descriptor is the lowest file descriptor not previously open for that process. No process can have more than **OPEN MAX** file descriptors open simultaneously.

The file offset, marking the current position within the file, is set to the beginning of the file. The new file descriptor is set to remain open across exec ("exec: execl, execle, execlp, execv, execve, execve, or exect Subroutine" on page 196) subroutines.

The open64 and creat64 subroutines are equivalent to the open and creat subroutines except that the O LARGEFILE flag is set in the open file description associated with the returned file descriptor. This flag allows files larger than OFF MAX to be accessed. If the caller attempts to open a file larger than OFF MAX and O LARGEFILE is not set, the open will fail and errno will be set to EOVERFLOW.

In the large file enabled programming environment, open is redefined to be open64 and creat is redefined to be creat64.

Parameters

Path Specifies the file to be opened.

Mode

Specifies the read, write, and execute permissions of the file to be created (requested by the O_CREAT flag). If the file already exists, this parameter is ignored. The Mode parameter is constructed by logically ORing one or more of the following values, which are defined in the sys/mode.h file:

S ISUID

Enables the setuid attribute for an executable file. A process executing this program acquires the access rights of the owner of the file.

S_ISGID

Enables the **setgid** attribute for an executable file. A process executing this program acquires the access rights of the group of the file. Also, enables the group-inheritance attribute for a directory. Files created in this directory have a group equal to the group of the directory.

The following attributes apply only to files that are directly executable. They have no meaning when applied to executable text files such as shell scripts and awk scripts.

S ISVTX

Enables the link/unlink attribute for a directory. Files cannot be linked to in this directory. Files can only be unlinked if the requesting process has write permission for the directory and is either the owner of the file or the directory.

S_ISVTX

Enables the save text attribute for an executable file. The program is not unmapped after

S_ENFMT

Enables enforcement-mode record locking for a regular file. File locks requested with the lockf subroutine are enforced.

S_IRUSR

Permits the file's owner to read it.

S_IWUSR

Permits the file's owner to write to it.

S IXUSR

Permits the file's owner to execute it (or to search the directory).

S IRGRP

Permits the file's group to read it.

S IWGRP

Permits the file's group to write to it.

S IXGRP

Permits the file's group to execute it (or to search the directory).

S IROTH

Permits others to read the file.

S IWOTH

Permits others to write to the file.

S IXOTH

Permits others to execute the file (or to search the directory).

Other mode values exist that can be set with the **mknod** subroutine but not with the chmod subroutine.

Extension

Provides communication with character device drivers that require additional information or return additional status. Each driver interprets the Extension parameter in a device-dependent way, either as a value or as a pointer to a communication area. Drivers must apply reasonable defaults when the Extension parameter value is 0.

OFlag

Specifies the type of access, special open processing, the type of update, and the initial state of the open file. The parameter value is constructed by logically ORing special open processing flags. These flags are defined in the **fcntl.h** file and are described in the following flags.

Flags That Specify Access Type

The following *OFlag* parameter flag values specify type of access:

O_RDONLY The file is opened for reading only.

O_WRONLY The file is opened for writing only.

O RDWR The file is opened for both reading and writing.

Note: One of the file access values must be specified. Do not use **O_RDONLY**, **O_WRONLY**, or **O_RDWR** together. If none is set, none is used, and the results are unpredictable.

Flags That Specify Special Open Processing

The following *OFlag* parameter flag values specify special open processing:

O_CREAT

If the file exists, this flag has no effect, except as noted under the **O_EXCL** flag. If the file does not exist, a regular file is created with the following characteristics:

- The owner ID of the file is set to the effective user ID of the process.
- The group ID of the file is set to the group ID of the parent directory if the parent directory has the SetGroupID attribute (S_ISGID bit) set. Otherwise, the group ID of the file is set to the effective group ID of the calling process.
- The file permission and attribute bits are set to the value of the Mode parameter, modified as follows:
 - All bits set in the process file mode creation mask are cleared. (The file creation mask is described in the umask subroutine.)
 - The S ISVTX attribute bit is cleared.

O_EXCL

If the O_EXCL and O_CREAT flags are set, the open is unsuccessful if the file exists.

Note: The **O_EXCL** flag is not fully supported for Network File Systems (NFS). The NFS protocol does not guarantee the designed function of the **O_EXCL** flag.

O_NSHARE

Assures that no process has this file open and precludes subsequent opens. If the file is on a physical file system and is already open, this open is unsuccessful and returns immediately unless the *OFlag* parameter also specifies the **O_DELAY** flag. This flag is effective only with physical file systems.

Note: This flag is not supported by NFS.

O_RSHARE

Assures that no process has this file open for writing and precludes subsequent opens for writing. The calling process can request write access. If the file is on a physical file system and is open for writing or open with the **O_NSHARE** flag, this open fails and returns immediately unless the *OFlag* parameter also specifies the **O_DELAY** flag.

Note: This flag is not supported by NFS.

O DEFER

The file is opened for deferred update. Changes to the file are not reflected on permanent storage until an **fsync** ("fsync or fsync_range Subroutine" on page 276) subroutine operation is performed. If no **fsync** subroutine operation is performed, the changes are discarded when the file is closed.

Note: This flag causes modified pages to be backed by paging space. Before using this flag make

sure there is sufficient paging space.

O_NOCTTY

This flag specifies that the controlling terminal should not be assigned during this open.

Note: This flag is not supported by NFS or JFS2, and the flag will be quietly ignored.

O TRUNC

If the file does not exist, this flag has no effect. If the file exists, is a regular file, and is successfully opened with the **O_RDWR** flag or the **O_WRONLY** flag, all of the following apply:

- · The length of the file is truncated to 0.
- · The owner and group of the file are unchanged.
- The SetUserID attribute of the file mode is cleared.
- · The SetUserID attribute of the file is cleared.

O_DIRECT

This flag specifies that direct i/o will be used for this file while it is opened.

O_CIO

This flag specifies that concurrent i/o (CIO) will be used for the file while it is opened. Because implementing concurrent readers and writers utilizes the direct I/O path (with more specific requirements to improve performance for running database on file system), this flag will override the O_DIRECT flag if the two options are specified at the same time. Currently, only JFS2 and namefs, which includes a selected subset of JFS2 files/directories, support CIO. The length of data to be read/written and the file offset must be page-aligned to be transferred as direct i/o with concurrent readers and writers.

The O_CIO flag is exclusive. If the file is opened in any other way (for example, using theO_DIO flag or opening the file normally), the open will fail. If the file is opened using the O_CIO and another process attempts to open the file another way, the open will fail. The O CIO flag also prevents the mmap subroutine and the shmat subroutine access to the file. The mmap subroutine and the shmat subroutine return EINVAL if they are used on a file that was opened using the O_CIO flag.

O_SNAPSHOT The file being opened contains a JFS2 snapshot. Subsequent read calls using this file descriptor will read the cooked snapshot rather than the raw snapshot blocks. A snapshot can only have one active open file descriptor for it.

The **open** subroutine is unsuccessful if any of the following conditions are true:

- The file supports enforced record locks and another process has locked a portion of the file.
- The file is on a physical file system and is already open with the O_RSHARE flag or the O_NSHARE flag.
- The file does not allow write access.
- · The file is already opened for deferred update.

Flag That Specifies Type of Update

A program can request some control on when updates should be made permanent for a regular file opened for write access. The following *OFlag* parameter values specify the type of update performed:

O SYNC:

If set, updates to regular files and writes to block devices are synchronous updates. File update is performed by the following subroutines:

- fclear
- ftruncate
- open with O_TRUNC
- write

On return from a subroutine that performs a synchronous update (any of the preceding subroutines, when the O SYNC flag is set), the program is assured that all data for the file has been written to permanent storage, even if the file is also open for deferred update.

Note: The **O_DSYNC** flag applies to AIX 4.2.1 and later releases.

O_DSYNC:

If set, the file data as well as all file system meta-data required to retrieve the file data are written to their permanent storage locations. File attributes such as access or modification times are not required to retrieve file data, and as such, they are not guaranteed to be written to their permanent storage locations before the preceding subroutines return. (Subroutines listed in the O_SYNC description.)

O SYNC | O DSYNC:

If both flags are set, the file's data and all of the file's meta-data (including access time) are written to their permanent storage locations.

Note: The **O_RSYNC** flag applies to AIX 4.3 and later releases.

O RSYNC: This flag is used in combination with O SYNC or D SYNC, and it extends

> their write operation behaviors to read operations. For example, when O_SYNC and R_SYNC are both set, a read operation will not return until the file's data and all of the file's meta-data (including access time) are written to

their permanent storage locations.

Flags That Define the Open File Initial State

The following *OFlag* parameter flag values define the initial state of the open file:

O_APPEND The file pointer is set to the end of the file prior to each write operation.

O DELAY Specifies that if the open subroutine could not succeed due to an inability to grant the access on

a physical file system required by the O_RSHARE flag or the O_NSHARE flag, the process

blocks instead of returning the ETXTBSY error code.

O_NDELAY Opens with no delay.

O_NONBLOCK Specifies that the **open** subroutine should not block.

The O_NDELAY flag and the O_NONBLOCK flag are identical except for the value returned by the read and write subroutines. These flags mean the process does not block on the state of an object, but does block on input or output to a regular file or block device.

The O_DELAY flag is relevant only when used with the O_NSHARE or O_RSHARE flags. It is unrelated to the O_NDELAY and O_NONBLOCK flags.

General Notes on OFlag Parameter Flags

The effect of the O CREAT flag is immediate, even if the file is opened with the O DEFER flag.

When opening a file on a physical file system with the O NSHARE flag or the O RSHARE flag, if the file is already open with conflicting access the following can occur:

- If the O_DELAY flag is clear (the default), the open subroutine is unsuccessful.
- If the O DELAY flag is set, the open subroutine blocks until there is no conflicting open. There is no deadlock detection for processes using the **O DELAY** flag.

When opening a file on a physical file system that has already been opened with the O_NSHARE flag, the following can occur:

- If the O_DELAY flag is clear (the default), the open is unsuccessful immediately.
- If the O DELAY flag is set, the open blocks until there is no conflicting open.

When opening a file with the O_RDWR, O_WRONLY, or O_TRUNC flag, and the file is already open with the **O_RSHARE** flag:

- If the O DELAY flag is clear (the default), the open is unsuccessful immediately.
- If the O DELAY flag is set, the open blocks until there is no conflicting open.

When opening a first-in-first-out (FIFO) with the **O RDONLY** flag, the following can occur:

- If the O NDELAY and O NONBLOCK flags are clear, the open blocks until a process opens the file for writing. If the file is already open for writing (even by the calling process), the **open** subroutine returns without delay.
- If the O NDELAY flag or the O NONBLOCK flag is set, the open succeeds immediately even if no process has the FIFO open for writing.

When opening a FIFO with the **O_WRONLY** flag, the following can occur:

- If the O NDELAY and O NONBLOCK flags are clear (the default), the open blocks until a process opens the file for reading. If the file is already open for writing (even by the calling process), the open subroutine returns without delay.
- If the O_NDELAY flag or the O_NONBLOCK flag is set, the open subroutine returns an error if no process currently has the file open for reading.

When opening a block special or character special file that supports nonblocking opens, such as a terminal device, the following can occur:

- If the O_NDELAY and O_NONBLOCK flags are clear (the default), the open blocks until the device is ready or available.
- If the O_NDELAY flag or the O_NONBLOCK flag is set, the open subroutine returns without waiting for the device to be ready or available. Subsequent behavior of the device is device-specific.

Any additional information on the effect, if any, of the O_NDELAY, O_RSHARE, O_NSHARE, and O DELAY flags on a specific device is documented in the description of the special file related to the device type.

If path refers to a STREAMS file, oflag may be constructed from O NONBLOCK OR-ed with either O RDONLY, O WRONLY or O RDWR. Other flag values are not applicable to STREAMS devices and have no effect on them. The value O NONBLOCK affects the operation of STREAMS drivers and certain functions applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of **O NONBLOCK** is device-specific.

If path names the master side of a pseudo-terminal device, then it is unspecified whether open locks the slave side so that it cannot be opened. Portable applications must call unlockpt before opening the slave side.

The largest value that can be represented correctly in an object of type off t will be established as the offset maximum in the open file description.

Return Values

Upon successful completion, the file descriptor, a nonnegative integer, is returned. Otherwise, a value of -1 is returned, no files are created or modified, and the errno global variable is set to indicate the error.

Error Codes

The open, openx, and creat subroutines are unsuccessful and the named file is not opened if one or more of the following are true:

EACCES One of the following is true:

- The file exists and the type of access specified by the *OFlag* parameter is denied.
- Search permission is denied on a component of the path prefix specified by the Path parameter. Access could be denied due to a secure mount.
- · The file does not exist and write permission is denied for the parent directory of the file to

• The **O_TRUNC** flag is specified and write permission is denied.

EAGAIN The O TRUNC flag is set and the named file contains a record lock owned by another

The directory in which the entry for the new link is being placed cannot be extended, or an **EDQUOT**

i-node could not be allocated for the file, because the user or group quota of disk blocks or

i-nodes in the file system containing the directory has been exhausted.

The O_CREAT and O_EXCL flags are set and the named file exists. **EEXIST**

EFBIG An attempt was made to write a file that exceeds the process' file limit or the maximum file

> size. If the user has set the environment variable XPG_SUS_ENV=ON prior to execution of the process, then the SIGXFSZ signal is posted to the process when exceeding the process'

file size limit.

A signal was caught during the open subroutine. **EINTR**

The path parameter names a STREAMS file and a hangup or error occurred. **EIO**

Named file is a directory and write access is required (the O WRONLY or O RDWR flag is **EISDIR**

set in the OFlag parameter).

EMFILE The system limit for open file descriptors per process has already been reached

(OPEN_MAX).

ENAMETOOLONG The length of the Path parameter exceeds the system limit (PATH_MAX); or a path-name

component is longer than NAME_MAX and _POSIX_NO_TRUNC is in effect.

ENFILE The system file table is full.

ENOMEM

The O CREAT flag is not set and the named file does not exist; or the O CREAT flag is not **ENOENT**

> set and either the path prefix does not exist or the Path parameter points to an empty string. The Path parameter names a STREAMS file and the system is unable to allocate resources.

ENOSPC The directory or file system that would contain the new file cannot be extended.

ENOSR The Path argument names a STREAMS-based file and the system is unable to allocate a

STREAM.

ENOTDIR A component of the path prefix specified by the Path component is not a directory.

ENXIO One of the following is true:

· Named file is a character special or block special file, and the device associated with this

special file does not exist.

· Named file is a multiplexed special file and either the channel number is outside of the

valid range or no more channels are available.

• The O DELAY flag or the O NONBLOCK flag is set, the named file is a FIFO, the

O_WRONLY flag is set, and no process has the file open for reading.

EOVERFLOW A file greater than one terabyte was opened on the 32-bit kernel in JFS2. The exact max size

> is specified in MAX FILESIZE and may be obtained using the pathconf system call. Any file larger than that cannot be opened on the 32-bit kernel, but can be created and opened on

the 64-bit kernel.

EROFS Named file resides on a read-only file system and write access is required (either the

O_WRONLY, O_RDWR, O_CREAT (if the file does not exist), or O_TRUNC flag is set in the

OFlag parameter).

ETXTBSY File is on a physical file system and is already open in a manner (with the O RSHARE or

O NSHARE flag) that precludes this open; or the O NSHARE or O RSHARE flag was requested with the O NDELAY flag set, and there is a conflicting open on a physical file

system.

Note: The **EOVERFLOW** error code applies to AIX 4.2 and later releases.

EOVERFLOW A call was made to open and creat and the file already existed and its size was larger than

OFF_MAX and the O_LARGEFILE flag was not set.

The open, openx, and creat subroutines are unsuccessful if one of the following are true:

EFAULT The Path parameter points outside of the allocated address space of the process.

EINVAL The value of the *OFlag* parameter is not valid.

ELOOP Too many symbolic links were encountered in translating the *Path* parameter.

ETXTBSY The file specified by the Path parameter is a pure procedure (shared text) file that is currently

executing, and the **O_WRONLY** or **O_RDWR** flag is set in the *OFlag* parameter.

Related Information

The **chmod** ("chmod or fchmod Subroutine" on page 124) subroutine, **close** ("close Subroutine" on page 141) subroutine, exec ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutine, fcntl, dup, or dup2 ("fcntl, dup, or dup2 Subroutine" on page 214) subroutine, fsync ("fsync or fsync_range Subroutine" on page 276) subroutine, ioctl ("ioctl, ioctlx, ioctl32, or ioctl32x Subroutine" on page 459) subroutine, lockfx ("lockfx, lockf, flock, or lockf64 Subroutine" on page 628) subroutine, Iseek ("Iseek, Ilseek or Iseek64 Subroutine" on page 646) subroutine, read subroutine, stat subroutine, umask subroutine, write subroutine.

The Input and Output Handling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

opendir, readdir, telldir, seekdir, rewinddir, or closedir Subroutine

Purpose

Performs operations on directories.

Library

Standard C Library (libc.a)

Syntax

```
#include <dirent.h>
DIR *opendir ( DirectoryName)
const char *DirectoryName;
struct dirent *readdir ( DirectoryPointer)
DIR *DirectoryPointer;
long int telldir(DirectoryPointer)
DIR *DirectoryPointer;
void seekdir(DirectoryPointer,Location)
DIR *DirectoryPointer;
long Location;
void rewinddir (DirectoryPointer)
DIR *DirectoryPointer;
int closedir (DirectoryPointer)
DIR *DirectoryPointer;
```

Description

Attention: Do not use the readdir subroutine in a multithreaded environment. See the multithread alternative in the **readdir r** subroutine article.

The **opendir** subroutine opens the directory designated by the *DirectoryName* parameter and associates a directory stream with it.

Note: An open directory must always be closed with the closedir subroutine to ensure that the next attempt to open that directory is successful.

The **opendir** subroutine also returns a pointer to identify the directory stream in subsequent operations. The null pointer is returned when the directory named by the *DirectoryName* parameter cannot be accessed or when not enough memory is available to hold the entire stream. A successful call to any of the **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) functions closes any directory streams opened in the calling process.

The **readdir** subroutine returns a pointer to the next directory entry. The **readdir** subroutine returns entries for . (dot) and .. (dot dot), if present, but never returns an invalid entry (with d ino set to 0). When it reaches the end of the directory, or when it detects an invalid seekdir operation, the readdir subroutine returns the null value. The returned pointer designates data that may be overwritten by another call to the readdir subroutine on the same directory stream. A call to the readdir subroutine on a different directory stream does not overwrite this data. The **readdir** subroutine marks the st atime field of the directory for update each time the directory is actually read.

The telldir subroutine returns the current location associated with the specified directory stream.

The **seekdir** subroutine sets the position of the next **readdir** subroutine operation on the directory stream. An attempt to seek an invalid location causes the readdir subroutine to return the null value the next time it is called. The position should be that returned by a previous telldir subroutine call.

The rewinddir subroutine resets the position of the specified directory stream to the beginning of the directory.

The closedir subroutine closes a directory stream and frees the structure associated with the DirectoryPointer parameter.

If you use the fork ("fork, f_fork, or vfork Subroutine" on page 248) subroutine to create a new process from an existing one, either the parent or the child (but not both) may continue processing the directory stream using the readdir, rewinddir, or seekdir subroutine.

Parameters

DirectoryName Names the directory.

DirectoryPointer Points to the **DIR** structure of an open directory.

Location Specifies the offset of an entry relative to the start of the directory.

Return Values

On successful completion, the opendir subroutine returns a pointer to an object of type DIR. Otherwise, a null value is returned and the errno global variable is set to indicate the error.

On successful completion, the **readdir** subroutine returns a pointer to an object of type **struct dirent**. Otherwise, a null value is returned and the errno global variable is set to indicate the error. When the end of the directory is encountered, a null value is returned and the errno global variable is not changed by this function call.

On successful completion, the closedir subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

If the opendir subroutine is unsuccessful, it returns a null value and sets the errno global variable to one of the following values:

EACCES Indicates that search permission is denied for any component of the DirectoryName

parameter, or read permission is denied for the *DirectoryName* parameter.

ENAMETOOLONG Indicates that the length of the *DirectoryName* parameter argument exceeds the **PATH MAX**

value, or a path-name component is longer than the NAME MAX value while the

POSIX_NO_TRUNC value is in effect.

ENOENT Indicates that the named directory does not exist.

ENOTDIR Indicates that a component of the *DirectoryName* parameter is not a directory. **EMFILE** Indicates that too many file descriptors are currently open for the process.

If the readdir subroutine is unsuccessful, it returns a null value and sets the errno global variable to the following value:

EBADF Indicates that the *DirectoryPointer* parameter argument does not refer to an open directory stream.

If the closedir subroutine is unsuccessful, it returns a value of -1 and sets the errno global variable to the following value:

EBADF Indicates that the *DirectoryPointer* parameter argument does not refer to an open directory stream.

Examples

To search a directory for the entry name:

```
len = strlen(name);
DirectoryPointer = opendir(".");
for (dp = readdir(DirectoryPointer); dp != NULL; dp =
readdir(DirectoryPointer))
        if (dp->d namlen == len && !strcmp(dp->d name, name)) {
                closedir(DirectoryPointer);
                return FOUND:
closedir(DirectoryPointer);
return NOT FOUND;
```

Related Information

The **close** ("close Subroutine" on page 141) subroutine, **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutines, fork ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, **Iseek** ("Iseek, Ilseek or Iseek64 Subroutine" on page 646) subroutine, **openx**, **open**, or creat ("open, openx, open64, creat, or creat64 Subroutine" on page 763) subroutine, read, readv, readx, or readvx subroutine, scandir or alphasort subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

passwdexpired Subroutine

Purpose

Checks the user's password to determine if it has expired.

Syntax

```
passwdexpired ( UserName, Message)
char *UserName;
char **Message;
```

Description

The passwdexpired subroutine checks a user's password to determine if it has expired. The subroutine checks the registry variable in the /etc/security/user file to ascertain where the user is administered. If the registry variable is not defined, the passwdexpired subroutine checks the local, NIS, and DCE databases for the user definition and expiration time.

The passwdexpired subroutine may pass back informational messages, such as how many days remain until password expiration.

Parameters

UserName Specifies the user's name whose password is to be checked.

Message Points to a pointer that the passwdexpired subroutine allocates memory for and fills in. This string is

suitable for printing and issues messages, such as in how many days the password will expire.

Return Values

Upon successful completion, the passwdexpired subroutine returns a value of 0. If this subroutine fails, it returns one of the following values:

1 Indicates that the password is expired, and the user must change it.

2 Indicates that the password is expired, and only a system administrator may change it.

-1 Indicates that an internal error has occurred, such as a memory allocation (malloc) failure or database corruption.

Error Codes

The passwdexpired subroutine fails if one or more of the following values is true:

ENOENT Indicates that the user could not be found.

EPERM Indicates that the user did not have permission to check password expiration.

ENOMEM Indicates that memory allocation (malloc) failed. EINVAL Indicates that the parameters are not valid.

Related Information

The authenticate ("authenticate Subroutine" on page 94) subroutine.

The login command.

pathconf or fpathconf Subroutine

Purpose

Retrieves file-implementation characteristics.

Library

Standard C Library (libc.a)

Syntax

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

```
long pathconf ( Path, Name)
const char *Path;
int Name:
```

```
long fpathconf( FileDescriptor, Name)
int FileDescriptor, Name;
```

Description

The pathconf subroutine allows an application to determine the characteristics of operations supported by the file system contained by the file named by the Path parameter. Read, write, or execute permission of the named file is not required, but all directories in the path leading to the file must be searchable.

The **fpathconf** subroutine allows an application to retrieve the same information for an open file.

Parameters

Path Specifies the path name.

Specifies an open file descriptor. FileDescriptor

Name

Specifies the configuration attribute to be queried. If this attribute is not applicable to the file specified by the Path or FileDescriptor parameter, the pathconf subroutine returns an error. Symbolic values for the *Name* parameter are defined in the **unistd.h** file:

_PC_LINK_MAX

Specifies the maximum number of links to the file.

PC_MAX_CANON

Specifies the maximum number of bytes in a canonical input line. This value is applicable only to terminal devices.

PC_MAX_INPUT

Specifies the maximum number of bytes allowed in an input queue. This value is applicable only to terminal devices.

PC NAME MAX

Specifies the maximum number of bytes in a file name, not including a terminating null character. This number can range from 14 through 255. This value is applicable only to a directory file.

PC_PATH_MAX

Specifies the maximum number of bytes in a path name, including a terminating null character.

PC PIPE BUF

Specifies the maximum number of bytes guaranteed to be written atomically. This value is applicable only to a first-in-first-out (FIFO).

PC_CHOWN_RESTRICTED

Returns 0 if the use of the chown subroutine is restricted to a process with appropriate privileges, and if the chown subroutine is restricted to changing the group ID of a file only to the effective group ID of the process or to one of its supplementary group IDs.

PC NO TRUNC

Returns 0 if long component names are truncated. This value is applicable only to a directory file.

_PC_VDISABLE

This is always 0. No disabling character is defined. This value is applicable only to a terminal device.

_PC_AIX_DISK_PARTITION

Determines the physical partition size of the disk.

Note: The _PC_AIX_DISK_PARTITION variable is available only to the root user.

_PC_AIX_DISK_SIZE

Determines the disk size in megabytes.

Note: The _PC_AIX_DISK_SIZE variable is available only to the root user.

Note: The _PC_FILESIZEBITS and PC_SYNC_IO flags apply to AIX 4.3 and later releases.

_PC_FILESIZEBITS

Returns the minimum number of bits required to hold the file system's maximum file size as a signed integer. The smallest value returned is 32.

_PC_SYNC_IO

Returns -1 if the file system does not support the Synchronized Input and Output option. Any value other than -1 is returned if the file system supports the option.

Notes:

- 1. If the Name parameter has a value of _PC_LINK_MAX, and if the Path or FileDescriptor parameter refers to a directory, the value returned applies to the directory itself.
- 2. If the Name parameter has a value of _PC_NAME_MAX or _PC_NO_TRUNC, and if the Path or FileDescriptor parameter refers to a directory, the value returned applies to filenames within the directory.

- 3. If the Name parameter has a value if **PC PATH MAX**, and if the Path or FileDescriptor parameter refers to a directory that is the working directory, the value returned is the maximum length of a relative pathname.
- 4. If the *Name* parameter has a value of **PC PIPE BUF**, and if the *Path* parameter refers to a FIFO special file or the FileDescriptor parameter refers to a pipe or a FIFO special file, the value returned applies to the referenced object. If the Path or FileDescriptor parameter refers to a directory, the value returned applies to any FIFO special file that exists or can be created within the directory.
- 5. If the Name parameter has a value of _PC_CHOWN_RESTRICTED, and if the Path or FileDescriptor parameter refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.

Return Values

If the pathconf or fpathconf subroutine is successful, the specified parameter is returned. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error. If the variable corresponding to the Name parameter has no limit for the Path parameter or the FileDescriptor parameter. both the pathconf and fpathconf subroutines return a value of -1 without changing the errno global variable.

Error Codes

The **pathconf** or **fpathconf** subroutine fails if the following error occurs:

EINVAL The name parameter specifies an unknown or inapplicable characteristic.

The pathconf subroutine can also fail if any of the following errors occur:

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

EINVAL The implementation does not support an association of the Name parameter with the

specified file.

ENAMETOOLONG The length of the *Path* parameter string exceeds the **PATH_MAX** value.

ENAMETOOLONG Pathname resolution of a symbolic link produced an intermediate result whose length

exceeds PATH MAX.

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

ENOTDIR A component of the path prefix is not a directory.

ELOOP Too many symbolic links were encountered in resolving path.

The **fpathconf** subroutine can fail if either of the following errors occur:

EBADF The File Descriptor parameter is not valid.

EINVAL The implementation does not support an association of the Name parameter with the specified file.

Related Information

The "chown, Ichown, Ichown, chownx, or fchownx Subroutine" on page 127, "confstr Subroutine" on page 147, **sysconf** subroutine.

Files, Directories, and File Systems for Programmers, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pause Subroutine

Purpose

Suspends a process until a signal is received.

Library

Standard C Library (libc.a)

Syntax

#include <unistd.h>
int pause (void)

Description

The **pause** subroutine suspends the calling process until it receives a signal. The signal must not be one that is ignored by the calling process. The **pause** subroutine does not affect the action taken upon the receipt of a signal.

Return Values

If the signal received causes the calling process to end, the pause subroutine does not return.

If the signal is caught by the calling process and control is returned from the signal-catching function, the calling process resumes execution from the point of suspension. The **pause** subroutine returns a value of -1 and sets the **errno** global variable to **EINTR**.

Related Information

The **incinterval**, **alarm**, or **settimer** ("getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine" on page 329)subroutine, **kill** or **killpg** ("kill or killpg Subroutine" on page 478) subroutine, **sigaction**, **sigvec**, or **signal** subroutine, **wait**, **waitpid**, or **wait3** subroutine.

pcap_close Subroutine

Purpose

Closes the open files related to the packet capture descriptor and frees the memory used by the packet capture descriptor.

Library

pcap Library (libpcap.a)

Syntax

#include <pcap.h>

void pcap close(pcap t * p);

Description

The **pcap_close** subroutine closes the files associated with the packet capture descriptor and deallocates resources. If the **pcap_open_offline** subroutine was previously called, the **pcap_close** subroutine closes the *savefile*, a previously saved packet capture data file. Or the **pcap_close** subroutine closes the packet capture device if the **pcap_open_live** subroutine was previously called.

Parameters

р

Points to a packet capture descriptor as returned by the **pcap_open_live** or the **pcap_open_offline** subroutine.

Related Information

The pcap_open_live ("pcap_open_live Subroutine" on page 791) subroutine, pcap_open_offline ("pcap_open_offline Subroutine" on page 792) subroutine.

pcap_compile Subroutine

Purpose

Compiles a filter expression into a filter program.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
```

```
int pcap_compile(pcap_t * p, struct bpf_ program *fp, char * str,
int optimize, bpf u int32 netmask);
```

Description

The pcap_compile subroutine is used to compile the string str into a filter program. This filter program will then be used to filter, or select, the desired packets.

Parameters

netmask	Specifies the <i>netmask</i> of the network device. The <i>netmask</i> can be obtained from the pcap_lookupnet subroutine.
optimize	Controls whether optimization on the resulting code is performed.
p	Points to a packet capture descriptor returned from the pcap_open_offline or the pcap_open_live subroutine.
program	Points to a bpf_program struct which will be filled in by the pcap_compile subroutine if the subroutine is successful.
str	Contains the filter expression.

Return Values

Upon successful completion, the pcap_compile subroutine returns 0, and the program parameter will hold the filter program. If pcap compile subroutine is unsuccessful, -1 is returned.

Related Information

The pcap geterr ("pcap geterr Subroutine" on page 785) subroutine, pcap lookupnet ("pcap lookupnet Subroutine" on page 787) subroutine, pcap_open_live ("pcap_open_live Subroutine" on page 791) subroutine, pcap open offline ("pcap open offline Subroutine" on page 792) subroutine, pcap perror ("pcap_perror Subroutine" on page 793) subroutine, pcap_setfilter ("pcap_setfilter Subroutine" on page 794) subroutine.

pcap_datalink Subroutine

Purpose

Obtains the link layer type for the packet capture device.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
int pcap_datalink(pcap_t * p);
```

Description

The **pcap_datalink** subroutine returns the link layer type of the packet capture device, for example, IFT_ETHER. This is useful in determining the size of the datalink header at the beginning of each packet that is read.

Parameters

р

Points to the packet capture descriptor as returned by the **pcap_open_live** or the **pcap_open_offline** subroutine.

Return Values

The **pcap_datalink** subroutine returns the link layer type.

Note: Only call this subroutine after successful calls to either the **pcap_open_live** or the **pcap_open_offline** subroutine. Never call the **pcap_datalink** subroutine after a call to **pcap_close** as unpredictable results will occur.

Related Information

The **pcap_close** ("pcap_close Subroutine" on page 778) subroutine, **pcap_open_live** ("pcap_open_live Subroutine" on page 791) subroutine, **pcap_open_offline** ("pcap_open_offline Subroutine" on page 792) subroutine.

pcap_dispatch Subroutine

Purpose

Collects and processes packets.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
int pcap_dispatch(pcap_t * p, int cnt, pcap_handler callback,
    u_char * user);
```

Description

The **pcap_dispatch** subroutine reads and processes packets. This subroutine can be called to read and process packets that are stored in a previously saved packet capture data file, known as the *savefile*. The subroutine can also read and process packets that are being captured live.

Notice that the third parameter, callback, is of the type pcap handler. This is a pointer to a user-provided subroutine with three parameters. Define this user-provided subroutine as follows:

```
void user routine(u char *user, struct pcap pkthdr *phdr, u char *pdata)
```

The parameter, user, is the user parameter that is passed into the pcap_dispatch subroutine. The parameter, phdr, is a pointer to the pcap_pkthdr structure which precedes each packet in the savefile. The parameter, pdata, points to the packet data. This allows users to define their own handling of packet capture data.

Parameters

callback

Points to a user-provided routine that will be called for each packet read. The user is responsible for providing a valid pointer, and that unpredictable results can occur if an invalid pointer is supplied.

Note: The pcap_dump subroutine can also be specified as the callback parameter. If this is done, the pcap_dump_open subroutine should be called first. The pointer to the pcap_dumper_t struct returned from the pcap_dump_open subroutine should be used as the *user* parameter to the **pcap_dispatch** subroutine. The following program fragment illustrates this use:

```
pcap dumper t *pd
pcap_t * p;
int rc = 0;
pd = pcap dump open(p, "/tmp/savefile");
rc = pcap_dispatch(p, 0 , pcap_dump, (u_char *) pd);
```

cnt

Specifies the maximum number of packets to process before returning. A cnt of -1 processes all the packets received in one buffer. A cnt of 0 processes all packets until an error occurs, EOF is reached, or the read times out (when doing live reads and a non-zero read timeout is specified).

Points to a packet capture descriptor returned from the pcap_open_offline or the pcap_open_live subroutine. This will be used to store packet data that is read in.

user

Specifies the first argument to pass into the callback routine.

Return Values

Upon successful completion, the pcap dispatch subroutine returns the number of packets read. If EOF is reached in a savefile, zero is returned. If the pcap dispatch subroutine is unsuccessful, -1 is returned. In this case, the pcap geterr or pcap perror subroutine can be used to get the error text.

Related Information

The pcap_dump ("pcap_dump Subroutine") subroutine, pcap_dump_close ("pcap_dump_close Subroutine" on page 782) subroutine, pcap_dump_open ("pcap_dump_open Subroutine" on page 783) subroutine, pcap_geterr ("pcap_geterr Subroutine" on page 785) subroutine, pcap_open_live ("pcap_open_live Subroutine" on page 791) subroutine, pcap_open_offline ("pcap_open_offline Subroutine" on page 792) subroutine, pcap_perror ("pcap_perror Subroutine" on page 793) subroutine.

pcap_dump Subroutine

Purpose

Writes packet capture data to a binary file.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
void pcap dump(u char * user, struct pcap pkthdr * h, u char * sp);
```

Description

The pcap_dump subroutine writes the packet capture data to a binary file. The packet header data, contained in h, will be written to the file pointed to by the user file pointer, followed by the packet data from sp. Up to h->caplen bytes of sp will be written.

The file that user points to (where the pcap_dump subroutine writes to) must be open. To open the file and retrieve its pointer, use the **pcap_dump_open** subroutine.

The calling arguments for the pcap_dump subroutine are suitable for use with pcap_dispatch subroutine and the pcap_loop subroutine. To retrieve this data, the pcap_open_offline subroutine can be invoked with the name of the file that *user* points to as its first parameter.

Parameters

h	Contains the packet header data that will be written to the packet capture date file, known as the <i>savefile</i> . This data will be written ahead of the rest of the packet data.
sp	Points to the packet data that is to be written to the savefile.
user	Specifies the <i>savefile</i> file pointer which is returned from the pcap_dump_open subroutine. It should be cast to a u char * when passed in.

Related Information

The pcap_dispatch ("pcap_dispatch Subroutine" on page 780) subroutine, pcap_dump_close ("pcap_dump_close Subroutine") subroutine, pcap_dump_open ("pcap_dump_open Subroutine" on page 783) subroutine, pcap loop ("pcap loop Subroutine" on page 788) subroutine, pcap open live ("pcap_open_live Subroutine" on page 791) subroutine, pcap_open_offline ("pcap_open_offline") Subroutine" on page 792) subroutine.

pcap dump close Subroutine

Purpose

Closes a packet capture data file, know as a savefile.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
void pcap_dump_close(pcap_dumper_t * p);
```

Description

The pcap_dump_close subroutine closes a packet capture data file, known as the savefile, that was opened using the **pcap_dump_open** subroutine.

Parameters

Points to a pcap_dumper_t, which is synonymous with a FILE *, the file pointer of a savefile.

Related Information

The pcap_dump_open ("pcap_dump_open Subroutine") subroutine.

pcap dump open Subroutine

Purpose

Opens or creates a file for writing packet capture data.

Library

pcap Library (libpcap.a)

Syntax

#include <pcap.h>

pcap_dumper_t *pcap_dump_open(pcap_t * p, char * fname);

Description

The pcap dump open subroutine opens or creates the packet capture data file, known as the savefile. This action is specified through the *fname* parameter. The subroutine then writes the required packet capture file header to the file. The pcap dump subroutine can then be called to write the packet capture data associated with the packet capture descriptor, p, into this file. The pcap dump open subroutine must be called before calling the pcap_dump subroutine.

Parameters

fname

р

Specifies the name of the file to open. A "-" indicates that standard output should be used instead of a file. Specifies a packet capture descriptor returned by the pcap_open_offline or the pcap_open_live subroutine.

Return Values

Upon successful completion, the pcap_dump_open subroutine returns a pointer to a the file that was opened or created. This pointer is a pointer to a pcap dumper t, which is synonymous with FILE *. See the pcap_dump ("pcap_dump Subroutine" on page 781), pcap_dispatch ("pcap_dispatch Subroutine" on page 780), or the pcap loop ("pcap loop Subroutine" on page 788) subroutine for an example of how to use pcap_dumper_t. If the pcap_dump_open subroutine is unsuccessful, Null is returned. Use the pcap geterr subroutine to obtain the specific error text.

Related Information

The pcap dispatch ("pcap dispatch Subroutine" on page 780) subroutine, pcap dump ("pcap dump Subroutine" on page 781) subroutine, pcap_dump_close ("pcap_dump_close Subroutine" on page 782) subroutine, pcap geterr ("pcap geterr Subroutine" on page 785) subroutine, pcap loop ("pcap loop Subroutine" on page 788) subroutine, pcap open live ("pcap open live Subroutine" on page 791) subroutine, pcap open offline ("pcap open offline Subroutine" on page 792) subroutine.

pcap_file Subroutine

Purpose

Obtains the file pointer to the savefile, a previously saved packed capture data file.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
FILE *pcap file(pcap t * p);
```

Description

The **pcap_file** subroutine returns the file pointer to the *savefile*. If there is no open *savefile*, 0 is returned. This subroutine should be called after a successful call to the **pcap_open_offline** subroutine and before any calls to the **pcap_close** subroutine.

Parameters

р

Points to a packet capture descriptor as returned by the **pcap_open_offline** subroutine.

Return Values

The pcap_file subroutine returns the file pointer to the savefile.

Related Information

The **pcap_close** ("pcap_close Subroutine" on page 778) subroutine, **pcap_open_offline** ("pcap_open_offline Subroutine" on page 792) subroutine.

pcap_fileno Subroutine

Purpose

Obtains the descriptor for the packet capture device.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
int pcap fileno(pcap t * p);
```

Description

The **pcap_fileno** subroutine returns the descriptor for the packet capture device. This subroutine should be called only after a successful call to the **pcap_open_live** subroutine and before any calls to the **pcap_close** subroutine.

Parameters

р

Points to a packet capture descriptor as returned by the pcap_open_live subroutine.

Return Values

The pcap fileno subroutine returns the descriptor for the packet capture device.

Related Information

The pcap_close ("pcap_close Subroutine" on page 778) subroutine, pcap_open_live ("pcap_open_live Subroutine" on page 791) subroutine.

pcap_geterr Subroutine

Purpose

Obtains the most recent pcap error message.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
char *pcap_geterr(pcap_t * p);
```

Description

The pcap_geterr subroutine returns the error text pertaining to the last pcap library error. This subroutine is useful in obtaining error text from those subroutines that do not return an error string. Since the pointer returned points to a memory space that will be reused by the pcap library subroutines, it is important to copy this message into a new buffer if the error text needs to be saved.

Parameters

Points to a packet capture descriptor as returned by the pcap_open_live or the pcap_open_offline subroutine.

Return Values

The pcap_geterr subroutine returns a pointer to the most recent error message from a pcap library subroutine. If there were no previous error messages, a string with 0 as the first byte is returned.

Related Information

The pcap_open_live ("pcap_open_live Subroutine" on page 791) subroutine, pcap_open_offline ("pcap_open_offline Subroutine" on page 792) subroutine, pcap_perror ("pcap_perror Subroutine" on page 793) subroutine, pcap_strerror ("pcap_strerror Subroutine" on page 796) subroutine.

pcap_is_swapped Subroutine

Purpose

Reports if the byte order of the previously saved packet capture data file, known as the *savefile* was swapped.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
int pcap is swapped(pcap t * p);
```

Description

The **pcap_is_swapped** subroutine returns 1 (True) if the current *savefile* uses a different byte order than the current system. This subroutine should be called after a successful call to the **pcap_open_offline** subroutine and before any calls to the **pcap_close** subroutine.

Parameters

р

Points to a packet capture descriptor as returned from the **pcap_open_offline** subroutine.

Return Values

1

If the byte order of the *savefile* is different from that of the current system.

0

If the byte order of the *savefile* is the same as that of the current system.

Related Information

The **pcap_close** ("pcap_close Subroutine" on page 778) subroutine, **pcap_open_offline** ("pcap_open_offline Subroutine" on page 792) subroutine.

pcap_lookupdev Subroutine

Purpose

Obtains the name of a network device on the system.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
char *pcap_lookupdev(char * errbuf);
```

Description

The pcap_lookupdev subroutine gets a network device suitable for use with the pcap_open_live and the pcap_lookupnet subroutines. If no interface can be found, or none are configured to be up, Null is returned. In the case of multiple network devices attached to the system, the pcap_lookupdev subroutine returns the first one it finds to be up, other than the loopback interface. (Loopback is always ignored.)

Parameters

errbuf

Returns error text and is only set when the pcap_lookupdev subroutine fails.

Return Values

Upon successful completion, the pcap lookupdev subroutine returns a pointer to the name of a network device attached to the system. If pcap_lookupdev subroutine is unsuccessful, Null is returned, and text indicating the specific error is written to errbuf.

Related Information

The pcap_geterr ("pcap_geterr Subroutine" on page 785) subroutine, pcap_lookupnet ("pcap_lookupnet Subroutine") subroutine, pcap_open_live ("pcap_open_live Subroutine" on page 791) subroutine, pcap_perror ("pcap_perror Subroutine" on page 793) subroutine.

pcap_lookupnet Subroutine

Purpose

Returns the network address and subnet mask for a network device.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
```

```
int pcap lookupnet(char * device, bpf u int32 * netp, bpf u int32 * maskp,
char * errbuf);
```

Description

Use the pcap_lookupnet subroutine to determine the network address and subnet mask for the network device, device.

Parameters

device	Specifies the name of the network device to use for the network lookup, for example, en0.
errbuf	Returns error text and is only set when the pcap_lookupnet subroutine fails.
maskp	Holds the subnet mask associated with device.
netp	Holds the network address for the device.

Return Values

Upon successful completion, the **pcap_lookupnet** subroutine returns 0. If the **pcap_lookupnet** subroutine is unsuccessful, -1 is returned, and *errbuf* is filled in with an appropriate error message.

Related Information

The **pcap_compile** ("pcap_compile Subroutine" on page 779) subroutine, **pcap_geterr** ("pcap_geterr Subroutine" on page 785) subroutine, **pcap_lookupdev** ("pcap_lookupdev Subroutine" on page 786) subroutine, **pcap_perror** ("pcap_perror Subroutine" on page 793) subroutine.

pcap_loop Subroutine

Purpose

Collects and processes packets.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
int pcap_loop(pcap_t * p, int cnt, pcap_handler callback,
    u char * user);
```

Description

The **pcap_loop** subroutine reads and processes packets. This subroutine can be called to read and process packets that are stored in a previously saved packet capture data file, known as the *savefile*. The subroutine can also read and process packets that are being captured live.

This subroutine is similar to **pcap_dispatch** subroutine except it continues to read packets until *cnt* packets have been processed, EOF is reached (in the case of offline reading), or an error occurs. It does not return when live read timeouts occur. That is, specifying a non-zero read timeout to the **pcap_open_live** subroutine and then calling the **pcap_loop** subroutine allows the reception and processing of any packets that arrive when the timeout occurs.

Notice that the third parameter, *callback*, is of the type **pcap_handler**. This is a pointer to a user-provided subroutine with three parameters. Define this user-provided subroutine as follows:

```
void user routine(u char *user, struct pcap pkthdr *phrd, u char *pdata)
```

The parameter, *user*, will be the user parameter that was passed into the **pcap_dispatch** subroutine. The parameter, *phdr*, is a pointer to the **pcap_pkthdr** structure, which precedes each packet in the *savefile*. The parameter, *pdata*, points to the packet data. This allows users to define their own handling of their filtered packets.

Parameters

callback

Points to a user-provided routine that will be called for each packet read. The user is responsible for providing a valid pointer, and that unpredictable results can occur if an invalid pointer is supplied.

Note: The pcap_dump subroutine can also be specified as the callback parameter. If this is done, call the pcap_dump_open subroutine first. Then use the pointer to the pcap_dumper_t struct returned from the pcap_dump_open subroutine as the user parameter to the pcap_dispatch subroutine. The following program fragment illustrates this use:

```
pcap dumper t *pd
pcap t * p;
int rc = 0:
pd = pcap_dump_open(p, "/tmp/savefile");
rc = pcap dispatch(p, 0 , pcap dump, (u char *) pd);
```

cnt

Specifies the maximum number of packets to process before returning. A negative value causes the pcap loop subroutine to loop forever, or until EOF is reached or an error occurs. A cnt of 0 processes all packets until an error occurs or EOF is reached. Points to a packet capture descriptor returned from the pcap_open_offline or the pcap_open_live subroutine. This will be used to store packet data that is read in.

р

user

Specifies the first argument to pass into the callback routine.

Return Values

Upon successful completion, the pcap loop subroutine returns 0.0 is also returned if EOF has been reached in a savefile. If the pcap loop subroutine is unsuccessful, -1 is returned. In this case, the pcap_geterr subroutine or the pcap_perror subroutine can be used to get the error text.

Related Information

The pcap_dispatch ("pcap_dispatch Subroutine" on page 780) subroutine, pcap_dump ("pcap_dump Subroutine" on page 781) subroutine, pcap dump close ("pcap dump close Subroutine" on page 782) subroutine, pcap_dump_open ("pcap_dump_open Subroutine" on page 783) subroutine, pcap_geterr ("pcap_geterr Subroutine" on page 785) subroutine, pcap_open_live ("pcap_open_live Subroutine" on page 791) subroutine, pcap_open_offline ("pcap_open_offline Subroutine" on page 792) subroutine, pcap perror ("pcap perror Subroutine" on page 793) subroutine.

pcap major version Subroutine

Purpose

Obtains the major version number of the packet capture format used to write the savefile, a previously saved packet capture data file.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
int pcap major version(pcap t * p);
```

Description

The **pcap_major_version** subroutine returns the major version number of the packet capture format used to write the *savefile*. If there is no open *savefile*, 0 is returned.

Note: This subroutine should be called only after a successful call to **pcap_open_offline** subroutine and before any calls to the **pcap close** subroutine.

Parameters

р

Points to a packet capture descriptor as returned from **pcap open offline** subroutine.

Return Values

The major version number of the packet capture format used to write the savefile.

Related Information

The **pcap_close** ("pcap_close Subroutine" on page 778) subroutine, **pcap_open_offline** ("pcap_open_offline Subroutine" on page 792) subroutine.

pcap_minor_version Subroutine

Purpose

Obtains the minor version number of the packet capture format used to write the savefile.

Library

pcap Library (libpcap.a)

Syntax

#include <pcap.h>

int pcap minor version(pcap t * p);

Description

The **pcap_minor_version** subroutine returns the minor version number of the packet capture format used to write the *savefile*. This subroutine should only be called after a successful call to the **pcap_open_offline** subroutine and before any calls to the **pcap_close** subroutine.

Parameters

р

Points to a packet capture descriptor as returned from the **pcap_open_offline** subroutine.

Return Values

The minor version number of the packet capture format used to write the savefile.

Related Information

The **pcap_close** ("pcap_close Subroutine" on page 778) subroutine, **pcap_open_offline** ("pcap_open_offline Subroutine" on page 792) subroutine.

pcap_next Subroutine

Purpose

Obtains the next packet from the packet capture device.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
```

```
u char *pcap next(pcap t * p, struct pcap pkthdr * h);
```

Description

The **pcap** next subroutine returns a u char pointer to the next packet from the packet capture device. The packet capture device can be a network device or a savefile that contains packet capture data. The data has the same format as used by tcpdump.

Parameters

h

Points to the packet header of the packet that is returned. This is filled in upon return by this routine.

р

Points to the packet capture descriptor to use as returned by the pcap_open_live or the pcap_open_offline subroutine.

Return Values

Upon successful completion, the pcap_next subroutine returns a pointer to a buffer containing the next packet and fills in the h, which points to the packet header of the returned packet. If the pcap next subroutine is unsuccessful. Null is returned.

Related Information

The pcap dispatch ("pcap dispatch Subroutine" on page 780) subroutine, pcap dump ("pcap dump Subroutine" on page 781) subroutine, pcap_dump_close ("pcap_dump_close Subroutine" on page 782) subroutine, pcap_dump_open ("pcap_dump_open Subroutine" on page 783) subroutine, pcap_loop ("pcap_loop Subroutine" on page 788) subroutine, pcap_open_live ("pcap_open_live Subroutine") subroutine, pcap open offline ("pcap open offline Subroutine" on page 792) subroutine.

The **tcpdump** command.

pcap_open_live Subroutine

Purpose

Opens a network device for packet capture.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
pcap t *pcap open live(char * device, int snaplen,
  int promisc, int to ms, char * ebuf);
```

Description

The pcap_open_live subroutine opens the specified network device for packet capture. The term "live" is to indicate that a network device is being opened, as opposed to a file that contains packet capture data. This subroutine must be called before any packet capturing can occur. All other routines dealing with packet capture require the packet capture descriptor that is created and initialized with this routine. See the pcap_open_offline ("pcap_open_offline Subroutine") subroutine for more details on opening a previously saved file that contains packet capture data.

Parameters

device	Specifies a string that contains the name of the network device to open for packet capture, for example, en0.
ebuf	Returns error text and is only set when the pcap_open_live subroutine fails.
promisc	Specifies that the device is to be put into promiscuous mode. A value of 1 (True) turns promiscuous mode on. If this parameter is 0 (False), the device will remain unchanged. In this case, if it has already been set to promiscuous mode (for some other reason), it will remain in this mode.
snaplen	Specifies the maximum number of bytes to capture per packet.
to_ms	Specifies the read timeout in milliseconds.

Return Values

Upon successful completion, the pcap_open_live subroutine will return a pointer to the packet capture descriptor that was created. If the pcap_open_live subroutine is unsuccessful, Null is returned, and text indicating the specific error is written into the ebuf buffer.

Related Information

The pcap_close ("pcap_close Subroutine" on page 778) subroutine, pcap_compile ("pcap_compile") Subroutine" on page 779) subroutine, pcap_datalink ("pcap_datalink Subroutine" on page 779) subroutine, pcap dispatch ("pcap dispatch Subroutine" on page 780) subroutine, pcap dump ("pcap_dump Subroutine" on page 781) subroutine, pcap_dump_open ("pcap_dump_open Subroutine" on page 783) subroutine, pcap geterr ("pcap geterr Subroutine" on page 785) subroutine, pcap loop ("pcap_loop Subroutine" on page 788) subroutine, pcap_open_offline ("pcap_open_offline Subroutine") subroutine, pcap perror ("pcap perror Subroutine" on page 793) subroutine, pcap setfilter ("pcap_setfilter Subroutine" on page 794) subroutine, pcap_snapshot ("pcap_setfilter Subroutine" on page 794) subroutine, pcap stats ("pcap stats Subroutine" on page 795) subroutine.

pcap_open_offline Subroutine

Purpose

Opens a previously saved file containing packet capture data.

Library

pcap Library (libpcap.a)

Syntax

#include <pcap.h>

pcap t *pcap open offline(char * fname, char * ebuf);

Description

The pcap open offline subroutine opens a previously saved packet capture data file, known as the savefile. This subroutine creates and initializes a packet capture (pcap) descriptor and opens the specified savefile containing the packet capture data for reading.

This subroutine should be called before any other related routines that require a packet capture descriptor for offline packet processing. See the pcap open live ("pcap open live Subroutine" on page 791) subroutine for more details on live packet capture.

Note: The format of the savefile is expected to be the same as the format used by the tcpdump command.

Parameters

ebuf

fname

Returns error text and is only set when the pcap_open_offline subroutine fails.

Specifies the name of the file to open. A hyphen (-) passed as the *fname* parameter indicates that stdin should

be used as the file to open.

Return Values

Upon successful completion, the pcap open offline subroutine will return a pointer to the newly created packet capture descriptor. If the pcap_open_offline subroutine is unsuccessful, Null is returned, and text indicating the specific error is written into the ebuf buffer.

Related Information

The pcap close ("pcap close Subroutine" on page 778) subroutine, pcap dispatch ("pcap dispatch Subroutine" on page 780) subroutine, pcap file ("pcap file Subroutine" on page 784) subroutine, pcap fileno ("pcap fileno Subroutine" on page 784) subroutine, pcap geterr ("pcap geterr Subroutine" on page 785) subroutine, pcap_is_swapped ("pcap_is_swapped Subroutine" on page 786) subroutine, pcap loop ("pcap loop Subroutine" on page 788) subroutine, pcap major version ("pcap major version Subroutine" on page 789) subroutine, pcap minor version ("pcap minor version Subroutine" on page 790) subroutine, pcap next ("pcap next Subroutine" on page 791) subroutine, pcap open live ("pcap_open_live Subroutine" on page 791) subroutine.

The **tcpdump** command.

pcap_perror Subroutine

Purpose

Prints the passed-in prefix, followed by the most recent error text.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
void pcap_perror(pcap_t * p, char * prefix);
```

Description

The **pcap_perror** subroutine prints the text of the last pcap library error to stderr, prefixed by *prefix*. If there were no previous errors, only *prefix* is printed.

Parameters

Points to a packet capture descriptor as returned by the

pcap_open_live subroutine or the pcap_open_offline subroutine.

prefix Subrout

Specifies the string that is to be printed before the stored

error message.

Related Information

The **pcap_geterr** ("pcap_geterr Subroutine" on page 785) subroutine, **pcap_open_live** ("pcap_open_live Subroutine" on page 791) subroutine, **pcap_open_offline** ("pcap_open_offline Subroutine" on page 792) subroutine, **pcap_strerror** ("pcap_strerror Subroutine" on page 796) subroutine.

pcap_setfilter Subroutine

Purpose

Loads a filter program into a packet capture device.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
```

```
int pcap_setfilter(pcap_t * p, struct bpf_program * fp);
```

Description

The **pcap_setfilter** subroutine is used to load a filter program into the packet capture device. This causes the capture of the packets defined by the filter to begin.

Parameters

fp Points to a filter program as returned from the

pcap_compile subroutine.

p Points to a packet capture descriptor returned from the

pcap open offline or the pcap open live subroutine.

Return Values

Upon successful completion, the pcap setfilter subroutine returns 0. If the pcap setfilter subroutine is unsuccessful, -1 is returned. In this case, the pcap_geterr subroutine can be used to get the error text, and the **pcap_perror** subroutine can be used to display the text.

Related Information

The pcap_compile ("pcap_compile Subroutine" on page 779) subroutine, pcap_geterr ("pcap_geterr Subroutine" on page 785) subroutine, pcap_open_live ("pcap_open_live Subroutine" on page 791) subroutine, pcap_open_offline ("pcap_open_offline Subroutine" on page 792) subroutine, pcap_perror ("pcap_perror Subroutine" on page 793) subroutine.

pcap_snapshot Subroutine

Purpose

Obtains the number of bytes that will be saved for each packet captured.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
int pcap snapshot( pcap t * p);
```

Description

The pcap_snapshot subroutine returns the snapshot length, which is the number of bytes to save for each packet captured.

Note: This subroutine should only be called after successful calls to either the pcap open live subroutine or pcap_open_offline subroutine. It should not be called after a call to the pcap_close subroutine.

Parameters

Points to the packet capture descriptor as returned by the pcap_open_live or the pcap_open_offline subroutine.

Return Values

The **pcap** snapshot subroutine returns the snapshot length.

Related Information

The pcap close ("pcap close Subroutine" on page 778) subroutine, pcap open live ("pcap open live Subroutine" on page 791) subroutine, pcap_open_offline ("pcap_open_offline Subroutine" on page 792) subroutine.

pcap_stats Subroutine

Purpose

Obtains packet capture statistics.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
int pcap_stats (pcap_t *p, struct pcap_stat *ps);
```

Description

The **pcap_stats** subroutine fills in a **pcap_stat** struct. The values represent packet statistics from the start of the run to the time of the call. Statistics for both packets that are received by the filter and packets that are dropped are stored inside a **pcap_stat** struct. This subroutine is for use when a packet capture device is opened using the **pcap_open_live** subroutine.

Parameters

р

ps

Points to a packet capture descriptor as returned by the **pcap open live** subroutine.

Points to the **pcap_stat** struct that will be filled in with the packet capture statistics.

Return Values

On successful completion, the **pcap_stats** subroutine fills in *ps* and returns 0. If the **pcap_stats** subroutine is unsuccessful, -1 is returned. In this case, the error text can be obtained with the **pcap_perror** subroutine or the **pcap_geterr** subroutine.

Related Information

The **pcap_geterr** ("pcap_geterr Subroutine" on page 785) subroutine, **pcap_open_live** ("pcap_open_live Subroutine" on page 791) subroutine, **pcap_perror** ("pcap_perror Subroutine" on page 793) subroutine.

pcap_strerror Subroutine

Purpose

Obtains the error message indexed by error.

Library

pcap Library (libpcap.a)

Syntax

```
#include <pcap.h>
char *pcap strerror(int error);
```

Description

Lookup the error message indexed by *error*. The possible values of *error* correspond to the values of the *errno* global variable. This function is equivalent to the **strerror** subroutine.

Parameters

error

Specifies the key to use in obtaining the corresponding error message. The error message is taken from the system's sys_errlist.

Return Values

The pcap_strerror subroutine returns the appropriate error message from the system error list.

Related Information

The pcap_geterr ("pcap_geterr Subroutine" on page 785) subroutine, pcap_perror ("pcap_perror Subroutine" on page 793) subroutine, **strerror** subroutine.

pclose Subroutine

Purpose

Closes a pipe to a process.

Library

Standard C Library (libc.a)

Syntax

#include <stdio.h> int pclose (Stream) **FILE** *Stream;

Description

The **pclose** subroutine closes a pipe between the calling program and a shell command to be executed. Use the pclose subroutine to close any stream you opened with the popen subroutine. The pclose subroutine waits for the associated process to end, and then returns the exit status of the command.

Attention: If the original processes and the **popen** process are reading or writing a common file, neither the popen subroutine nor the pclose subroutine should use buffered I/O. If they do, the results are unpredictable.

Avoid problems with an output filter by flushing the buffer with the **fflush** subroutine.

Parameter

Specifies the FILE pointer of an opened pipe. Stream

Return Values

The **pclose** subroutine returns a value of -1 if the *Stream* parameter is not associated with a **popen** command or if the status of the child process could not be obtained. Otherwise, the value of the termination status of the command language interpreter is returned; this will be 127 if the command language interpreter cannot be executed.

Error Codes

If the application has:

- Called the wait subroutine,
- · Called the waitpid subroutine with a process ID less than or equal to zero or equal to the process ID of the command line interpreter.
- · Masked the SIGCHILD signal, or
- · Called any other function that could perform one of the steps above, and

one of these calls caused the termination status to be unavailable to the pclose subroutine, a value of -1 is returned and the **errno** global variable is set to **ECHILD**.

Related Information

The fclose or fflush ("fclose or fflush Subroutine" on page 213) subroutine, fopen, freopen, or fdopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, pipe ("pipe Subroutine" on page 815) subroutine, popen ("popen Subroutine" on page 864) subroutine, wait, waitpid, or wait3 subroutine.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

perfstat_cpu Subroutine

Purpose

Retrieves individual logical CPU usage statistics.

Library

perfstat library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
int perfstat_cpu (name, userbuff, sizeof struct, desired number)
perfstat_id_t * name;
perfstat cpu t * userbuff;
size_t sizeof struct;
int desired number;
```

Description

The **perfstat cpu** subroutine retrieves one or more individual CPU usage statistics. The same function can be used to retrieve the number of available sets of logical CPU statistics.

To get one or more sets of CPU usage metrics, set the name parameter to the name of the first CPU for which statistics are desired, and set the desired number parameter. To start from the first CPU, set the name parameter to "". The userbuff parameter must always point to a memory area big enough to contain the desired number of perfstat cpu t structures that will be copied by this function. Upon return, the name parameter will be set to either the name of the next CPU, or to "" after all structures have been copied.

To retrieve the number of available sets of CPU usage metrics, set the *name* and *userbuff* parameters to NULL, and the desired number parameter to 0. The returned value will be the number of available sets.

This number represents the number of logical processors for which statistics are available. In a dynamic LPAR environment, this number is the highest logical index of an online processor since the last reboot. See the Perfstat API article in Performance Tools and APIs Technical Reference for more information on the **perfstat_cpu** subroutine and DLPAR.

Parameters

Contains either "", FIRST_CPU, or a name identifying the first logical CPU for which statistics are name

desired. Logical processor names are:

cpu0, cpu1,...

To provide binary compatibility with previous versions of the library, names like proc0, proc1, ... will still be accepted. These names will be treated as if their corresponding cpuN name was used,

but the names returned in the structures will always be names starting with cpu.

userbuff Points to the memory area that is to be filled with one or more **perfstat_cpu_t** structures.

sizeof_struct Specifies the size of the **perfstat_cpu_t** structure: sizeof(perfstat_cpu_t). Specifies the number of **perfstat_cpu_t** structures to copy to *userbuff*. desired_number

Return Values

Unless the **perfstat** cpu subroutine is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes

The **perfstat cpu** subroutine is unsuccessful if the following is true:

EINVAL One of the parameters is not valid.

Files

The libperfstat.h file defines standard macros, data types, and subroutines.

Related Information

"perfstat_netbuffer Subroutine" on page 807, "perfstat_cpu_total Subroutine," "perfstat_disk Subroutine" on page 801, "perfstat diskadapter Subroutine" on page 802, "perfstat diskpath Subroutine" on page 803, "perfstat disk total Subroutine" on page 805, "perfstat memory total Subroutine" on page 806, "perfstat_netinterface Subroutine" on page 808, "perfstat_netinterface_total Subroutine" on page 810, "perfstat_pagingspace Subroutine" on page 811, "perfstat_protocol Subroutine" on page 812, and "perfstat reset Subroutine" on page 814.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_cpu_total Subroutine

Purpose

Retrieves global CPU usage statistics.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
int perfstat cpu total (name, userbuff, sizeof struct, desired number)
perfstat id t *name;
perfstat_cpu_total_t *userbuff;
size t sizeof struct;
int desired_number;
```

Description

The perfstat_cpu_total subroutine returns global CPU usage statistics in a perfstat_cpu_total_t structure.

To get statistics that are global to the whole system, the name parameter must be set to NULL, the userbuff parameter must be allocated, and the desired number parameter must be set to 1.

The **perfstat cpu total** subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The perfstat reset subroutine must be called to flush the dictionary whenever the machine configuration has changed.

Parameters

name Must be set to NULL.

userbuff Points to the memory area that is to be filled with the **perfstat_cpu_total_t** structure. sizeof_struct Specifies the size of the perfstat_cpu_total_t structure: sizeof(perfstat cpu total t).

desired_number Must be set to 1.

Return Values

Upon successful completion, the number of structures filled is returned. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes

The **perfstat cpu total** subroutine is unsuccessful if one of the following is true:

EINVAL One of the parameters is not valid.

EFAULT Insufficient memory.

ENOMEM The string default length is too short.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

Related Information

"perfstat_netbuffer Subroutine" on page 807, "perfstat_cpu Subroutine" on page 798, "perfstat_disk Subroutine" on page 801, "perfstat_diskadapter Subroutine" on page 802, "perfstat_diskpath Subroutine" on page 803, "perfstat_disk_total Subroutine" on page 805, "perfstat_memory_total Subroutine" on page 806, "perfstat netinterface Subroutine" on page 808, "perfstat netinterface total Subroutine" on page 810, "perfstat pagingspace Subroutine" on page 811, and "perfstat protocol Subroutine" on page 812.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_disk Subroutine

Purpose

Retrieves individual disk usage statistics.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
int perfstat_disk (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat_disk_t *userbuff;
size_t sizeof_struct;
int desired number;
```

Description

The **perfstat_disk** subroutine retrieves one or more individual disk usage statistics. The same function can also be used to retrieve the number of available sets of disk statistics.

To get one or more sets of disk usage metrics, set the *name* parameter to the name of the first disk for which statistics are desired, and set the *desired_number* parameter. To start from the first disk, specify "" or FIRST_DISK as the *name*. The *userbuff* parameter must always point to a memory area big enough to contain the desired number of **perfstat_disk_t** structures that will be copied by this function. Upon return, the *name* parameter will be set to either the name of the next disk, or to "" after all structures have been copied.

To retrieve the number of available sets of disk usage metrics, set the *name* and *userbuff* parameters to NULL, and the *desired_number* parameter to 0. The returned value will be the number of available sets.

The **perfstat_disk** subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The **perfstat_reset** subroutine must be called to flush the dictionary whenever the machine configuration has changed.

Parameters

name Contains either "", FIRST_DISK, or a name identifying the first disk for which statistics are

desired. For example:

hdiskO, hdisk1, ...

userbuffPoints to the memory area to be filled with one or more perfstat_disk_t structures.sizeof_structSpecifies the size of the perfstat_disk_t structure: sizeof(perfstat_disk_t)desired_numberSpecifies the number of perfstat_disk_t structures to copy to userbuff.

Return Values

Unless the function is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_disk** subroutine is unsuccessful if one of the following is true:

EINVAL One of the parameters is not valid.

EFAULT Insufficient memory.

The string default length is too short. **ENOMEM**

ENOMSG Cannot access the dictionary.

Files

The libperfstat.h file defines standard macros, data types, and subroutines.

Related Information

"perfstat_netbuffer Subroutine" on page 807, "perfstat_cpu Subroutine" on page 798, "perfstat_cpu_total Subroutine" on page 799, "perfstat_diskadapter Subroutine," "perfstat_diskpath Subroutine" on page 803, "perfstat_disk_total Subroutine" on page 805, "perfstat_memory_total Subroutine" on page 806, "perfstat_netinterface Subroutine" on page 808, "perfstat_netinterface_total Subroutine" on page 810, "perfstat_pagingspace Subroutine" on page 811, "perfstat_protocol Subroutine" on page 812, and "perfstat reset Subroutine" on page 814.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat diskadapter Subroutine

Purpose

Retrieves individual disk adapter usage statistics.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_diskadapter (name, userbuff, sizeof struct, desired number)
perfstat_id_t *name;
perfstat diskadapter t *userbuff;
size t sizeof struct;
int desired number;
```

Description

The perfstat_diskadapter subroutine retrieves one or more individual disk adapter usage statistics. The same function can be used to retrieve the number of available sets of adapter statistics.

To get one or more sets of disk adapter usage metrics, set the *name* parameter to the name of the first disk adapter for which statistics are desired, and set the desired_number parameter. To start from the first disk adapter, set the name parameter to "" or FIRST_DISKADAPTER. The userbuff parameter must point to a memory area big enough to contain the desired number of perfstat_diskadapter_t structures which will be copied by this function. Upon return, the name parameter will be set to either the name of the next disk adapter, or to "" if all structures have been copied.

To retrieve the number of available sets of disk adapter usage metrics, set the name and userbuff parameters to NULL, and the desired_number parameter to 0. The returned value will be the number of available sets.

The **perfstat diskadapter** subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The perfstat reset subroutine must be called to flush the dictionary whenever the machine configuration has changed.

Parameters

name Contains either "", FIRST_DISKADAPTER, or a name identifying the first disk adapter for

which statistics are desired. For example:

scsi0, scsi1, ...

userbuff Points to the memory area to be filled with one or more perfstat diskadapter_t

structures.

Specifies the size of the **perfstat diskadapter t** structure: sizeof struct

sizeof(perfstat diskadapter t)

desired number Specifies the number of perfstat_diskadapter_t structures to copy to userbuff.

Return Values

Unless the function is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes

The **perfstat diskadapter** subroutine is unsuccessful if one of the following is true:

EINVAL One of the parameters is not valid.

EFAULT Insufficient memory.

ENOMEM The string default length is too short.

ENOMSG Cannot access the dictionary.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

Related Information

"perfstat netbuffer Subroutine" on page 807, "perfstat cpu Subroutine" on page 798, "perfstat cpu total Subroutine" on page 799, "perfstat_disk Subroutine" on page 801, "perfstat_diskpath Subroutine," "perfstat_disk_total Subroutine" on page 805, "perfstat_memory_total Subroutine" on page 806, "perfstat netinterface Subroutine" on page 808, "perfstat netinterface total Subroutine" on page 810, "perfstat pagingspace Subroutine" on page 811, "perfstat protocol Subroutine" on page 812, and "perfstat reset Subroutine" on page 814.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_diskpath Subroutine

Purpose

Retrieves individual disk path usage statistics.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
int perfstat diskpath (name, userbuff, sizeof struct, desired number)
perfstat id t *name;
perfstat diskpath t *userbuff
size_t sizeof_struct;
int desired number;
```

Description

The perfstat_diskpath subroutine retrieves one or more individual disk path usage statistics. The same function can also be used to retrieve the number of available sets of disk path statistics.

To get one or more sets of disk path usage metrics, set the name parameter to the name of the first disk path for which statistics are desired, and set the desired_number parameter. To start from the first disk path, specify "" or FIRST DISKPATH as the *name* parameter. To start from the first path of a specific disk, set the *name* parameter to the diskname. The *userbuff* parameter must always point to a memory area big enough to contain the desired number of perfstat diskpath t structures that will be copied by this function. Upon return, the *name* parameter will be set to either the name of the next disk path, or to "" after all structures have been copied.

To retrieve the number of available sets of disk path usage metrics, set the name and userbuff parameters to NULL, and the *desired number* parameter to 0. The number of available sets is returned.

The perfstat diskpath subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The perfstat_reset subroutine must be called to flush the dictionary whenever the machine configuration has changed.

Parameters

Contains either "", FIRST_DISKPATH, a name identifying the first disk path for which statistics name

are desired, or a name identifying a disk for which path statistics are desired. For example:

hdisk0 Path2, hdisk1 Path0, ... or hdisk5 (equivalent to hdisk5 Pathfirstpath)

userbuff Points to the memory area to be filled with one or more perfstat_diskpath_t structures. sizeof_struct Specifies the size of the **perfstat diskpath t** structure: sizeof(perfstat diskpath t)

Specifies the number of perfstat_diskpath_t structures to copy to userbuff. desired_number

Return Values

Unless the function is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes

The perfstat diskpath subroutine is unsuccessful if one of the following is true:

EINVAL One of the parameters is not valid.

EFAULT Insufficient memory.

ENOMEM The string default length is too short. **ENOMSG** Cannot access the dictionary.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

Related Information

"perfstat netbuffer Subroutine" on page 807, "perfstat_cpu Subroutine" on page 798, "perfstat_cpu_total Subroutine" on page 799, "perfstat_disk Subroutine" on page 801, "perfstat_diskadapter Subroutine" on page 802, "perfstat_diskpath Subroutine" on page 803, "perfstat_disk_total Subroutine," "perfstat_memory_total Subroutine" on page 806, "perfstat_netinterface Subroutine" on page 808, "perfstat_netinterface_total Subroutine" on page 810, "perfstat_pagingspace Subroutine" on page 811, "perfstat_protocol Subroutine" on page 812, and "perfstat_reset Subroutine" on page 814.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat disk total Subroutine

Purpose

Retrieves global disk usage statistics.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
int perfstat disk total (name, userbuff, sizeof struct, desired number)
perfstat_id_t *name;
perfstat disk total t *userbuff;
size_t sizeof_struct;
int desired number;
```

Description

The perfstat_disk_total subroutine returns global disk usage statistics in a perfstat_disk_total_t structure.

To get statistics that are global to the whole system, the name parameter must be set to NULL, the userbuff parameter must be allocated, and the desired_number parameter must be set to 1.

The perfstat disk total subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The perfstat reset subroutine must be called to flush the dictionary whenever the machine configuration has changed.

Parameters

name Must be set to NULL.

Points to the memory area that is to be filled with one or more perfstat_disk_total_t userbuff

sizeof_struct Specifies the size of the **perfstat_disk_total_t** structure: sizeof(perfstat disk total t)

desired_number Must be set to 1.

Return Values

Upon successful completion, the number of structures that could be filled is returned. This will always be 1. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The perfstat_disk_total subroutine is unsuccessful if one of the following is true:

EINVAL One of the parameters is not valid.

EFAULT Insufficient memory.

ENOMEM The string default length is too short.

Files

The libperfstat.h file defines standard macros, data types, and subroutines.

Related Information

"perfstat_netbuffer Subroutine" on page 807, "perfstat_cpu Subroutine" on page 798, "perfstat_cpu_total Subroutine" on page 799, "perfstat_disk Subroutine" on page 801, "perfstat_diskadapter Subroutine" on page 802, "perfstat_diskpath Subroutine" on page 803, "perfstat_memory_total Subroutine," "perfstat_netinterface Subroutine" on page 808, "perfstat_netinterface_total Subroutine" on page 810, "perfstat_pagingspace Subroutine" on page 811, "perfstat_protocol Subroutine" on page 812, and "perfstat_reset Subroutine" on page 814.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_memory_total Subroutine

Purpose

Retrieves global memory usage statistics.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
int perfstat memory total (name, userbuff, sizeof struct, desired number)
perfstat id t *name;
perfstat_memory_total_t *userbuff;
size t sizeof struct;
int desired_number;
```

Description

The perfstat_memory_total subroutine returns global memory usage statistics in a perfstat_memory_total_t structure.

To get statistics that are global to the whole system, the *name* parameter must be set to NULL, the userbuff parameter must be allocated, and the desired number parameter must be set to 1.

Parameters

name Must be set to NULL.

userbuff Points to the memory area that is to be filled with the **perfstat_memory_total_t** structure.

Specifies the size of the **perfstat_memory_total_t** structure: sizeof_struct

sizeof(perfstat memory total t).

desired_number Must be set to 1.

Return Values

Upon successful completion, the number of structures filled is returned. This will always be 1. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes

The **perfstat memory total** subroutine is unsuccessful if the following is true:

EINVAL One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

Related Information

"perfstat_netbuffer Subroutine," "perfstat_cpu Subroutine" on page 798, "perfstat_cpu_total Subroutine" on page 799, "perfstat_disk Subroutine" on page 801, "perfstat_diskadapter Subroutine" on page 802, "perfstat_diskpath Subroutine" on page 803, "perfstat_disk_total Subroutine" on page 805, "perfstat netinterface Subroutine" on page 808, "perfstat netinterface total Subroutine" on page 810, "perfstat pagingspace Subroutine" on page 811, and "perfstat protocol Subroutine" on page 812.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_netbuffer Subroutine

Purpose

Retrieves network buffer allocation usage statistics.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
int perfstat_netbuffer (name, userbuff, sizeof struct, desired number)
perfstat id t *name;
perfstat netbuffer t *userbuff;
size_t sizeof struct;
int desired number;
```

Description

The perfstat netbuffer subroutine retrieves statistics about network buffer allocations for each possible buffer size. Returned counts are the sum of allocation statistics for all processors (kernel statistics are kept per size per processor) corresponding to a buffer size.

To get one or more sets of network buffer allocation usage metrics, set the *name* parameter to the network buffer size for which statistics are desired, and set the desired_number parameter. To start from the first network buffer size, specify "" or FIRST_NETBUFFER in the name parameter. The userbuff parameter must point to a memory area big enough to contain the desired number of perfstat_netbuffer_t structures which will be copied by this function.

Upon return, the name parameter will be set to either the ASCII size of the next buffer type, or to "" if all structures have been copied. Only the statistics for network buffer sizes that have been used are returned. Consequently, there can be holes in the returned array of statistics, and the structure corresponding to allocations of size 4096 may directly follow the structure for size 256 (in case 512, 1024 and 2048 have not been used yet). The structure corresponding to a buffer size not used yet is returned (with all fields set to 0) when it is directly asked for by name.

To retrieve the number of available sets of network buffer usage metrics, set the name and userbuff parameters to NULL, and the desired_number parameter to 0. The returned value will be the number of available sets.

Parameters

Contains either "", FIRST_NETBUFFER, or the size of the network buffer in ASCII. It is a name

power of 2. For example:

32, 64, 128, ..., 16384

userbuff Points to the memory area to be filled with one or more perfstat_netbuffer_t structures. sizeof_struct Specifies the size of the **perfstat netbuffer t** structure: sizeof(perfstat netbuffer t)

desired_number Specifies the number of perfstat_netbuffer_t structures to copy to userbuff.

Return Values

Upon successful completion, the number of structures which could be filled is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat netbuffer** subroutine is unsuccessful if the following is true:

EINVAL One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

Related Information

"perfstat_cpu Subroutine" on page 798, "perfstat_cpu_total Subroutine" on page 799, "perfstat memory total Subroutine" on page 806, "perfstat disk Subroutine" on page 801, "perfstat_diskpath Subroutine" on page 803, "perfstat_disk_total Subroutine" on page 805, "perfstat_netinterface_total Subroutine" on page 810, "perfstat_diskadapter Subroutine" on page 802, "perfstat_protocol Subroutine" on page 812, and "perfstat_pagingspace Subroutine" on page 811.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_netinterface Subroutine

Purpose

Retrieves individual network interface usage statistics.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
int perfstat netinterface (name, userbuff, sizeof struct, desired number)
perfstat id t *name;
perfstat_netinterface_t *userbuff;
size t sizeof struct;
int desired_number;
```

Description

The **perfstat netinterface** subroutine retrieves one or more individual network interface usage statistics. The same function can also be used to retrieve the number of available sets of network interface statistics.

To get one or more sets of network interface usage metrics, set the name parameter to the name of the first network interface for which statistics are desired, and set the desired_number parameter. To start from the first network interface, set the name parameter to "" or FIRST NETINTERFACE. The userbuff parameter must always point to a memory area big enough to contain the desired number of perfstat netinterface t structures that will be copied by this function. Upon return, the name parameter will be set to either the name of the next network interface, or to "" after all structures have been copied.

To retrieve the number of available sets of network interface usage metrics, set the name and userbuff parameters to NULL, and the desired number parameter to 0. The returned value will be the number of available sets.

The perfstat_netinterface subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The perfstat reset subroutine must be called to flush the dictionary whenever the machine configuration has changed.

Parameters

Contains either "", FIRST_NETINTERFACE, or a name identifying the first network interface for name

which statistics are desired. For example;

en0, tr10, ...

userbuff Points to the memory area that is to be filled with one or more perfstat netinterface t

structures.

Specifies the size of the perfstat_netinterface_t structure: sizeof(perfstat netinterface t) sizeof_struct

Specifies the number of **perfstat_netinterface_t** structures to copy to *userbuff*. desired_number

Return Values

Upon successful completion unless the function is used to retrieve the number of available structures, the number of structures filled is returned. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes

The **perfstat netinterface** subroutine is unsuccessful if one of the following is true:

One of the parameters is not valid. EINVAL

EFAULT Insufficient memory.

ENOMEM The string default length is too short. **ENOMSG** Cannot access the dictionary.

Files

The libperfstat.h file defines standard macros, data types, and subroutines.

Related Information

"perfstat netbuffer Subroutine" on page 807, "perfstat cpu Subroutine" on page 798, "perfstat cpu total Subroutine" on page 799, "perfstat_disk Subroutine" on page 801, "perfstat_diskadapter Subroutine" on page 802, "perfstat_diskpath Subroutine" on page 803, "perfstat_disk_total Subroutine" on page 805, "perfstat_memory_total Subroutine" on page 806, "perfstat_netinterface_total Subroutine," "perfstat pagingspace Subroutine" on page 811, "perfstat protocol Subroutine" on page 812, and "perfstat reset Subroutine" on page 814.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat netinterface total Subroutine

Purpose

Retrieves global network interface usage statistics.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
int perfstat_netinterface_total (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat netinterface total t *userbuff;
size t sizeof struct;
int desired number;
```

Description

The perfstat_netinterface_total subroutine returns global network interface usage statistics in a perfstat netinterface total t structure.

To get statistics that are global to the whole system, the *name* parameter must be set to NULL, the userbuff parameter must be allocated, and the desired number parameter must be set to 1.

The perfstat_netinterface_total subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The perfstat reset subroutine must be called to flush the dictionary whenever the machine configuration has changed.

Parameters

Must be set to NULL. name

userbuff Points to the memory area that is to be filled with the perfstat netinterface total t structure.

Specifies the size of the **perfstat_netinterface_total_t** structure: sizeof_struct

sizeof(perfstat_netinterface_total_t).

Must be set to 1. desired_number

Return Values

Upon successful completion, the number of structures filled is returned. This will always be 1. If unsuccessful, a value of -1 is returned and the errno variable is set.

Error Codes

The perfstat_netinterface_total subroutine is unsuccessful if one of the following is true:

EINVAL One of the parameters is not valid.

EFAULT Insufficient memory.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

Related Information

"perfstat_netbuffer Subroutine" on page 807, "perfstat_cpu Subroutine" on page 798, "perfstat_cpu_total Subroutine" on page 799, "perfstat_disk Subroutine" on page 801, "perfstat_diskadapter Subroutine" on page 802, "perfstat_diskpath Subroutine" on page 803, "perfstat_disk_total Subroutine" on page 805, "perfstat memory total Subroutine" on page 806, "perfstat netinterface Subroutine" on page 808, "perfstat_pagingspace Subroutine," "perfstat_protocol Subroutine" on page 812, and "perfstat_reset Subroutine" on page 814.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_pagingspace Subroutine

Purpose

Retrieves individual paging space usage statistics.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_pagingspace (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat pagingspace t *userbuff;
size t sizeof struct;
int desired number;
```

Description

This function retrieves one or more individual pagingspace usage statistics. The same functions can also be used to retrieve the number of available sets of paging space statistics.

To get one or more sets of paging space usage metrics, set the *name* parameter to the name of the first paging space for which statistics are desired, and set the desired_number parameter. To start from the first paging space, set the name parameter to "" or FIRST_PAGINGSPACE. In either case, userbuff must point to a memory area big enough to contain the desired number of perfstat_pagingspace_t structures which will be copied by this function. Upon return, the name parameter will be set to either the name of the next paging space, or to "" if all structures have been copied.

To retrieve the number of available sets of paging space usage metrics, set the name and userbuff parameters to NULL, and the desired number parameter to 0. The number of available sets will be returned.

The perfstat_pagingspace subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The perfstat reset subroutine must be called to flush the dictionary whenever the machine configuration has changed.

Parameters

Contains either "", FIRST_PAGINGSPACE, or a name identifying the first paging space for name

which statistics are desired. For example:

paging00, hd6, ...

userbuff Points to the memory area to be filled with one or more perfstat_pagingspace_t structures.

Specifies the size of the **perfstat_pagingspace_t** structure: sizeof struct

sizeof(perfstat pagingspace t)

desired number Specifies the number of perfstat pagingspace t structures to copy to userbuff.

Return Values

Unless the perfstat_pagingspacesubroutine is used to retrieve the number of available structures, the number of structures which could be filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes

The **perfstat_pagingspace** subroutine is unsuccessful if one of the following are true:

EINVAL One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

Related Information

"perfstat_netbuffer Subroutine" on page 807, "perfstat_cpu Subroutine" on page 798, "perfstat_cpu_total Subroutine" on page 799, "perfstat_disk Subroutine" on page 801, "perfstat_diskadapter Subroutine" on page 802, "perfstat_diskpath Subroutine" on page 803, "perfstat_disk_total Subroutine" on page 805, "perfstat memory total Subroutine" on page 806, "perfstat netinterface Subroutine" on page 808, "perfstat_netinterface_total Subroutine" on page 810, "perfstat_protocol Subroutine," and "perfstat_reset Subroutine" on page 814.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_protocol Subroutine

Purpose

Retrieves protocol usage statistics.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_protocol (name, userbuff, sizeof struct, desired number)
perfstat id t *name;
perfstat_protocol_t *userbuff;
size t sizeof struct;
int desired number;
```

Description

The **perfstat_protocol** subroutine retrieves protocol usage statistics such as ICMP, ICMPv6, IP, IPv6, TCP, UDP, RPC, NFS, NFSv2, NFSv3. To get one or more sets of protocol usage metrics, set the *name* parameter to the name of the first protocol for which statistics are desired, and set the desired number parameter.

To start from the first protocol, set the name parameter to "" or FIRST_PROTOCOL. The userbuff parameter must point to a memory area big enough to contain the desired number of perfstat protocol t structures which will be copied by this function. Upon return, the *name* parameter will be set to either the name of the next protocol, or to "" if all structures have been copied.

To retrieve the number of available sets of protocol usage metrics, set the *name* and *userbuff* parameters to NULL, and the desired_number parameter to 0. The returned value will be the number of available sets.

Parameters

Contains either "ip", "ipv6", "icmp", "icmpv6", "tcp", "udp", "rpc", "nfs", "nfsv2", "nfsv3", "", or name

FIRST_PROTOCOL.

userbuff Points to the memory area to be filled with one or more perfstat_protocol_t structures. Specifies the size of the **perfstat protocol t** structure: sizeof(perfstat protocol t) sizeof_struct

desired number Specifies the number of **perfstat_protocol_t** structures to copy to *userbuff*.

Return Values

Upon successful completion, the number of structures which could be filled is returned. If unsuccessful, a value of -1 is returned and the errno global variable is set.

Error Codes

The **perfstat_protocol** subroutine is unsuccessful if the following is true:

EINVAL One of the parameters is not valid.

Files

The libperfstat.h file defines standard macros, data types, and subroutines.

Related Information

"perfstat_netbuffer Subroutine" on page 807, "perfstat_cpu Subroutine" on page 798, "perfstat_cpu_total Subroutine" on page 799, "perfstat_disk Subroutine" on page 801, "perfstat_diskadapter Subroutine" on page 802, "perfstat_diskpath Subroutine" on page 803, "perfstat_disk_total Subroutine" on page 805, "perfstat memory total Subroutine" on page 806, "perfstat netinterface Subroutine" on page 808, "perfstat netinterface total Subroutine" on page 810, and "perfstat pagingspace Subroutine" on page 811.

Perfstat API in Performance Tools and APIs Technical Reference.

perfstat_reset Subroutine

Purpose

Empties libperfstat configuration information cache.

Library

Perfstat Library (libperfstat.a)

Syntax

```
#include <libperfstat.h>
void perfstat_reset (void)
```

Description

The perfstat_cpu_total, perfstat_disk, perfstat_diskadapter, perfstat_netinterface, and perfstat_pagingspace subroutines return configuration information retrieved from the ODM database and automatically cached by the library.

The **perfstat_reset** subroutine flushes this information cache and should be called whenever the machine configuration has changed.

Files

The **libperfstat.h** defines standard macros, data types and subroutines.

Related Information

"perfstat_cpu_total Subroutine" on page 799, "perfstat_disk Subroutine" on page 801, "perfstat_diskadapter Subroutine" on page 802, "perfstat_diskpath Subroutine" on page 803, "perfstat_netinterface Subroutine" on page 808, and "perfstat_pagingspace Subroutine" on page 811.

Perfstat API in Performance Tools and APIs Technical Reference.

perror Subroutine

Purpose

Writes a message explaining a subroutine error.

Library

Standard C Library (libc.a)

Syntax

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

void perror ( String)
const char *String;
extern int errno;
extern char *sys_errlist[];
extern int sys_nerr;
```

Description

The **perror** subroutine writes a message on the standard error output that describes the last error encountered by a system call or library subroutine. The error message includes the String parameter string followed by a: (colon), a space character, the message, and a new-line character. The String parameter string should include the name of the program that caused the error. The error number is taken from the errno global variable, which is set when an error occurs but is not cleared when a successful call to the perror subroutine is made.

To simplify various message formats, an array of message strings is provided in the sys_errlist structure or use the errno global variable as an index into the sys_errlist structure to get the message string without the new-line character. The largest message number provided in the table is sys nerr. Be sure to check the sys nerr structure because new error codes can be added to the system before they are added to the table.

The **perror** subroutine retrieves an error message based on the language of the current locale.

After successfully completing, and before a call to the exit or abort subroutine or the completion of the fflush or fclose subroutine on the standard error stream, the perror subroutine marks for update the st ctime and st mtime fields of the file associated with the standard error stream.

Parameter

String

Specifies a parameter string that contains the name of the program that caused the error. The ensuing printed message contains this string, a: (colon), and an explanation of the error.

Related Information

The abort subroutine, exit subroutine, fflush or fclose subroutine, printf, fprintf, sprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf subroutine, strerror subroutine.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pipe Subroutine

Purpose

Creates an interprocess channel.

Library

Standard C Library (libc.a)

Syntax

#include <unistd.h>

int pipe (FileDescriptor) int FileDescriptor[2];

Description

The **pipe** subroutine creates an interprocess channel called a pipe and returns two file descriptors, FileDescriptor[0] and FileDescriptor[1]. FileDescriptor[0] is opened for reading and FileDescriptor[1] is opened for writing.

A read operation on the FileDescriptor[0] parameter accesses the data written to the FileDescriptor[1] parameter on a first-in, first-out (FIFO) basis.

Write requests of PIPE_BUF bytes or fewer will not be interleaved (mixed) with data from other processes doing writes on the same pipe. PIPE_BUF is a system variable described in the pathconf ("pathconf or fpathconf Subroutine" on page 774) subroutine. Writes of greater than PIPE BUF bytes may have data interleaved, on arbitrary boundaries, with other writes.

If O_NONBLOCK or O_NDELAY are set, writes requests of PIPE_BUF bytes or fewer will either succeed completely or fail and return -1 with the errno global variable set to EAGAIN. A write request for more than PIPE BUF bytes will either transfer what it can and return the number of bytes actually written, or transfer no data and return -1 with the errno global variable set to EAGAIN.

Parameters

FileDescriptor Specifies the address of an array of two integers into which the new file descriptors are

placed.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and the errno global variable is set to identify the error.

Error Codes

The **pipe** subroutine is unsuccessful if one or more the following are true:

EFAULT The FileDescriptor parameter points to a location outside of the allocated address space of the process.

EMFILE The number of open of file descriptors exceeds the **OPEN_MAX** value.

ENFILE The system file table is full, or the device containing pipes has no free i-nodes.

Related Information

The **read** subroutine, **select** subroutine, **write** subroutine.

The **ksh** command, **sh** command.

Files, Directories, and File Systems for Programmers in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

plock Subroutine

Purpose

Locks the process, text, or data in memory.

Library

Standard C Library (libc.a)

Syntax

#include <sys/lock.h>

int plock (Operation)

int Operation;

Description

The **plock** subroutine allows the calling process to lock or unlock its text region (text lock), its data region (data lock), or both its text and data regions (process lock) into memory. The plock subroutine does not lock the shared text segment or any shared data segments. Locked segments are pinned in memory and are immune to all routine paging. Memory locked by a parent process is not inherited by the children after a fork subroutine call. Likewise, locked memory is unlocked if a process executes one of the exec subroutines. The calling process must have the root user authority to use this subroutine.

A real-time process can use this subroutine to ensure that its code, data, and stack are always resident in memory.

Note: Before calling the plock subroutine, the user application must lower the maximum stack limit value using the ulimit subroutine.

Parameters

Operation Specifies one of the following:

PROCLOCK

Locks text and data into memory (process lock).

TXTLOCK

Locks text into memory (text lock).

DATLOCK

Locks data into memory (data lock).

UNLOCK

Removes locks.

Return Values

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **plock** subroutine is unsuccessful if one or more of the following is true:

EPERM The effective user ID of the calling process does not have the root user authority.

EINVAL The Operation parameter has a value other than PROCLOCK, TXTLOCK, DATLOCK, or UNLOCK. The Operation parameter is equal to PROCLOCK, and a process lock, text lock, or data lock already **EINVAL** exists on the calling process.

EINVAL The Operation parameter is equal to TXTLOCK, and a text lock or process lock already exists on the calling process.

EINVAL The Operation parameter is equal to DATLOCK, and a data lock or process lock already exists on the calling process.

EINVAL The Operation parameter is equal to UNLOCK, and no type of lock exists on the calling process.

Related Information

The **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutines, _exit, exit, or atexit ("exit, atexit, _exit, or _Exit Subroutine" on page 203)subroutine, fork ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, **ulimit** subroutine.

pm_cycles Subroutine

Purpose

Measures processor speed in cycles per second.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
double pm_cycles (void)
```

Description

The **pm_cycles** subroutine uses the Performance Monitor cycle counter and the processor real-time clock to measure the actual processor clock speed. The speed is returned in cycles per second.

Return Values

0 Processor speed in cycles per second

An error occurred.

No errors occurred.

Files

/usr/include/pmapi.h

Defines standard macros, data types, and subroutines.

Related Information

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_delete_program Subroutine

Purpose

Deletes previously established systemwide Performance Monitor settings.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_delete_program ()
```

Description

The **pm_delete_program** subroutine deletes previously established systemwide Performance Monitor settings.

Return Values

0 No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm error** ("pm error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

pm_init ("pm_init Subroutine" on page 837), pm_error ("pm_error Subroutine" on page 823), pm set program ("pm set program Subroutine" on page 844), pm get program ("pm get program Subroutine" on page 831), pm get data ("pm get data, pm get tdata, pm get data cpu, and pm get tdata cpu Subroutine" on page 824), pm start ("pm start Subroutine" on page 851), pm stop ("pm stop Subroutine" on page 856), pm reset data ("pm reset data Subroutine" on page 839) subroutines.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_delete_program_group Subroutine

Purpose

Deletes previously established Performance Monitor settings for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm delete_program_group ( pid, tid)
pid_t pid;
tid_t tid;
```

Description

The pm delete program group subroutine deletes previously established Performance Monitor settings for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of the calling process. The settings for the group to which the target thread belongs and from all the other threads in the same group are also deleted.

Parameters

pid Process identifier of target thread. The target process

must be a debuggee under the control of the calling

process.

tid Thread identifier of a target thread.

Return Values

0 No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_set_program_group ("pm_set_program_group Subroutine" on page 845) subroutine, pm_get_program_group ("pm_get_program_group Subroutine" on page 832) subroutine, pm_get_data_group ("pm_get_data_group and pm_get_tdata_group Subroutine" on page 825) subroutine, pm_start_group ("pm_start_group Subroutine" on page 852) subroutine, pm_stop_group ("pm_stop_group Subroutine" on page 857) subroutine, pm_reset_data_group ("pm_reset_data_group Subroutine" on page 840) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_delete_program_mygroup Subroutine

Purpose

Deletes previously established Performance Monitor settings for the counting group to which the calling thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_delete_program_mygroup ()

Description

The **pm_delete_program_mygroup** subroutine deletes previously established Performance Monitor settings for the calling kernel thread, the counting group to which it belongs, and for all the threads that are members of the same group.

Return Values

No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm error** ("pm error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

pm_init ("pm_init Subroutine" on page 837), pm_error ("pm_error Subroutine" on page 823), pm set program mygroup ("pm set program mygroup Subroutine" on page 847), pm_get_program_mygroup ("pm_get_program_mygroup Subroutine" on page 833), pm_get_data_mygroup ("pm get data mygroup or pm get tdata mygroup Subroutine" on page 827), pm start mygroup ("pm_start_mygroup Subroutine" on page 853), pm_stop_mygroup ("pm_stop_mygroup Subroutine" on page 858), pm reset data mygroup ("pm reset data mygroup Subroutine" on page 841) subroutines.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_delete_program_mythread Subroutine

Purpose

Deletes the previously established Performance Monitor settings for the calling thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_delete_program_mythread ()

Description

The pm_delete_program_mythread subroutine deletes the previously established Performance Monitor settings for the calling kernel thread.

Return Values

No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

pm_init ("pm_init Subroutine" on page 837), pm_error ("pm_error Subroutine" on page 823), pm set program mythread ("pm set program mythread Subroutine" on page 848), pm_get_program_mythread ("pm_get_program_mythread Subroutine" on page 835), pm get data mythread ("pm get data mythread or pm get tdata mythread Subroutine" on page 828), pm start mythread ("pm start mythread Subroutine" on page 854), pm stop mythread ("pm stop mythread Subroutine" on page 859), pm reset data mythread ("pm reset data mythread Subroutine" on page 842) subroutines.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_delete_program_thread Subroutine

Purpose

Deletes the previously established Performance Monitor settings for a target thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm delete program thread (pid, tid)
pid t pid;
tid t tid;
```

Description

The pm delete program thread subroutine deletes the previously established Performance Monitor settings for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of the calling process.

Parameters

Process identifier of target thread. Target process must be pid a debuggee under the control of the calling process.

tid Thread identifier of the target thread.

Return Values

No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

pm_init ("pm_init Subroutine" on page 837), pm_error ("pm_error Subroutine"), pm_set_program_thread ("pm_set_program_thread Subroutine" on page 850), pm_get_program_thread ("pm_get_program_thread Subroutine" on page 836), pm_get_data_thread ("pm_get_data_thread or pm_get_tdata_thread Subroutine" on page 829), pm_start_thread ("pm_start_thread Subroutine" on page 855), pm_stop_thread ("pm stop thread Subroutine" on page 860), pm reset data thread ("pm reset data thread Subroutine" on page 843) subroutines.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_error Subroutine

Purpose

Decodes Performance Monitor APIs error codes.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
void pm_error ( *Where, errorcode)
char *Where;
int errorcode:
```

Description

The pm_error subroutine writes a message on the standard error output that describes the parameter errorcode encountered by a Performance Monitor API library subroutine. The error message includes the Where parameter string followed by a : (colon), a space character, the message, and a new-line character. The Where parameter string includes the name of the program that caused the error.

Parameters

*Where Specifies where the error was encountered.

Specifies the error code as returned by one of the Performance Monitor APIs library errorcode

subroutines.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm init subroutine, pm set program subroutine, pm get program subroutine, pm_delete_program subroutine, pm_get_data subroutine, pm_start subroutine, pm_stop subroutine, pm_reset_data subroutine.

The pm_set_program_mythread subroutine, pm_get_program_mythread subroutine, pm_delete_program_mythread subroutine, pm_get_data_mythread subroutine, pm_start_mythread subroutine, pm_stop_mythread subroutine, pm_reset_data_mythread subroutine.

The pm_set_program_mygroup subroutine, pm_get_program_mygroup subroutine, pm_delete_program_mygroup subroutine, pm_get_data_mygroup subroutine, pm_start_mygroup subroutine, pm_stop_mygroup subroutine, pm_reset_data_mygroup subroutine.

The pm_set_program_thread subroutine, pm_get_program_thread subroutine, pm_delete_program_thread subroutine, pm_get_data_thread subroutine, pm_start_thread subroutine, pm_stop_thread subroutine, pm_reset_data_thread subroutine.

The pm_set_program_group subroutine, pm_get_program_group subroutine, pm_delete_program_group subroutine, pm_get_data_group subroutine, pm_start_group subroutine, pm_stop_group subroutine, pm_reset_data_group subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_get_data, pm_get_tdata, pm_get_data_cpu, and pm_get_tdata_cpu Subroutine

Purpose

Returns systemwide Performance Monitor data.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_get_data ( *pmdata)
pm_data_t *pmdata;
int pm_get_tdata (pmdata, * time)
pm_data_t *pmdata;
timebasestruct_t *time;
int pm_get_data_cpu (cpuid, *pmdata)
int cpuid;
pm_data_t *pmdata;
int pm_get_tdata_cpu (cpuid, *pmdata, *time)
int cpuid;
pm_data_t *pmdata;
timebasestruct_t *time;
```

Description

The pm_get_data subroutine retrieves the current systemwide Performance Monitor data.

The **pm_get_tdata** subroutine retrieves the current systemwide Performance Monitor data, and a timestamp indicating the last time the hardware counters were read.

The **pm_get_data_cpu** subroutine retrieves the current Performance Monitor data for the specified processor.

The pm get tdata cpu subroutine retrieves the current Performance Monitor data for the specified processor, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always a set (one per hardware counter on the machines used) of 64-bit values.

Parameters

Pointer to a structure that contains the returned systemwide Performance *pmdata

Monitor data.

Pointer to a structure containing the timebase value the last time the *time

hardware Performance Monitoring counters were read. This can be converted

to time using the time_base_to_time subroutine.

Logical processor identifier. cpuid

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_set_program ("pm_set_program Subroutine" on page 844) subroutine, pm_get_program ("pm_get_program Subroutine" on page 831) subroutine, pm_delete_program ("pm_delete_program Subroutine" on page 818) subroutine, pm start ("pm start Subroutine" on page 851) subroutine, pm stop ("pm stop Subroutine" on page 856) subroutine, pm reset data ("pm reset data Subroutine" on page 839) subroutine.

read real time or time base to time Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_get_data_group and pm_get_tdata_group Subroutine

Purpose

Returns Performance Monitor data for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_get_data_group (pid, tid, *pmdata)
pid t pid;
tid t tid;
pm_data_t *pmdata;
int pm_get_tdata_group (pid, tid, *pmdata, *time)
pm_data_t *pmdata;
pid_t pid;
tid t tid;
timebasestruct t *time;
```

Description

The pm get data group subroutine retrieves the current Performance Monitor data for the counting group to which a target kernel thread belongs. The thread must be stopped and must be part of a debuggee process under the control of the calling process.

The pm get tdata group subroutine retrieves the current Performance Monitor data for the counting group to which a target thread belongs, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, if it is a process level group, and if its counters are consistent with the sum of the counters for all of the threads in the group.

Parameters

pid Process identifier of a target thread. The target process

must be an argument of a debug process.

tid Thread identifier of a target thread.

*pmdata Pointer to a structure to return the Performance Monitor data for the group to which the target thread belongs. Pointer to a structure containing the timebase value the *time last time the hardware Performance Monitoring counters

were read. This can be converted to time using the

time_base_to_time subroutine.

Return Values

No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the pm error ("pm error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_set_program_group ("pm_set_program_group Subroutine" on page 845) subroutine, pm get program group ("pm get program group Subroutine" on page 832) subroutine, pm_get_data_group ("pm_get_data_group and pm_get_tdata_group Subroutine" on page 825) subroutine, pm start group ("pm start group Subroutine" on page 852) subroutine, pm stop group ("pm stop group Subroutine" on page 857) subroutine, pm reset data group ("pm reset data group Subroutine" on page 840) subroutine.

read real time or time base to time Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm get data mygroup or pm get tdata mygroup Subroutine

Purpose

Returns Performance Monitor data for the counting group to which the calling thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_get_data_mygroup (*pmdata)
pm_data_t *pmdata;
int pm_get_tdata_mygroup (*pmdata, *time)
pm_data_t *pmdata;
timebasestruct t *time;
```

Description

The pm get data mygroup subroutine retrieves the current Performance Monitor data for the group to which the calling kernel thread belongs.

The pm get tdata mygroup subroutine retrieves the current Performance Monitor data for the group to which the calling thread belongs, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, if it is a process level group, and if its counters are consistent with the sum of the counters for all of the threads in the group.

Parameters

*pmdata

*time

Pointer to a structure to return the Performance Monitor data for the group to which the calling thread belongs. Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the time_base_to_time subroutine.

Return Values

No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h

Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_set_program_mygroup ("pm_set_program_mygroup Subroutine" on page 847) subroutine, pm_get_program_mygroup ("pm_get_program_mygroup Subroutine" on page 833) subroutine, pm_get_data_mygroup ("pm_get_data_mygroup or pm_get_tdata_mygroup Subroutine" on page 827) subroutine, pm_start_mygroup ("pm_start_mygroup Subroutine" on page 853) subroutine, pm_stop_mygroup ("pm_stop_mygroup Subroutine" on page 858) subroutine, pm_reset_data_mygroup ("pm_reset_data_mygroup Subroutine" on page 841) subroutine.

read_real_time or time_base_to_time Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_get_data_mythread or pm_get_tdata_mythread Subroutine

Purpose

Returns Performance Monitor data for the calling thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_get_data_mythread (*pmdata)
pm_data_t *pmdata;
int pm_get_tdata_mythread (*pmdata, *time)
pm_data_t *pmdata;
timebasestruct t *time;
```

Description

The pm_get_data_mythread subroutine retrieves the current Performance Monitor data for the calling kernel thread.

The **pm_get_tdata_mythread** subroutine retrieves the current Performance Monitor data for the calling kernel thread, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values.

Parameters

*pmdata

Pointer to a structure to contain the returned Performance Monitor data for the calling kernel thread.

*time

Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the time_base_to_time subroutine.

Return Values

No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the pm error ("pm error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

pm_init ("pm_init Subroutine" on page 837), pm_error ("pm_error Subroutine" on page 823), pm_set_program_mythread ("pm_set_program_mythread Subroutine" on page 848), pm_get_program_mythread ("pm_get_program_mythread Subroutine" on page 835), pm get data mythread ("pm get data mythread or pm get tdata mythread Subroutine" on page 828), pm start mythread ("pm start mythread Subroutine" on page 854), pm stop mythread ("pm stop mythread Subroutine" on page 859), pm reset data mythread ("pm reset data mythread Subroutine" on page 842) subroutines.

read real time or time base to time Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm get data thread or pm get tdata thread Subroutine

Purpose

Returns Performance Monitor data for a target thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm get data thread (pid, tid, *pmdata)
pid_t pid;
tid t tid;
pm data t *pmdata;
int pm get tdata thread (pid, tid, *pmdata, *time)
```

```
pid_t pid;
tid t tid;
pm data t *pmdata;
timebasestruct t *time;
```

Description

The pm get data thread subroutine retrieves the current Performance Monitor data for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of a calling process.

The pm get tdata thread subroutine retrieves the current Performance Monitor data for a target thread, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values.

Parameters

*pmdata

pid Process identifier of a target thread. The target process

must be a debuggee of the caller process.

tid Thread identifier of a target thread.

Pointer to a structure to return the Performance Monitor

data for the target kernel thread.

*time Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters

were read. This can be converted to time using the

time base to time subroutine.

Return Values

No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

pm_init ("pm_init Subroutine" on page 837), pm_error ("pm_error Subroutine" on page 823), pm_set_program_thread ("pm_set_program_thread Subroutine" on page 850), pm_get_program_thread ("pm_get_program_thread Subroutine" on page 836), pm_get_data_thread ("pm_get_data_thread or pm_get_tdata_thread Subroutine" on page 829), pm_start_thread ("pm_start_thread Subroutine" on page 855), pm_stop_thread ("pm_stop_thread Subroutine" on page 860), pm_reset_data_thread ("pm_reset_data_thread Subroutine" on page 843) subroutines.

read_real_time or time_base_to_time Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_get_program Subroutine

Purpose

Retrieves systemwide Performance Monitor settings.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_get_program ( *prog)
pm_prog_t *prog;
```

Description

The pm get program subroutine retrieves the current systemwide Performance Monitor settings. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the pm_init subroutine.

The counting mode includes user mode, the kernel mode, the current counting state, and the process tree mode. If the process tree mode is on, the counting applies only to the calling process and its decendants.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a threshold value is also returned.

If the events are represented by a group ID, then the is_group bit is set in the mode, and the first element of the events array contains the group ID. The other elements of the events array are not meaningful.

Parameters

proq

Returns which Performance Monitor events and modes are set. Supported modes are:

PM USER

Counting processes running in user mode

PM KERNEL

Counting processes running in kernel mode

PM COUNT

Counting is on

PM PROCTREE

Counting applies only to the calling process and its descendants

Return Values

No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h

Defines standard macros, data types, and subroutines.

Related Information

pm init ("pm init Subroutine" on page 837), pm error ("pm error Subroutine" on page 823), pm set program ("pm set program Subroutine" on page 844), pm delete program ("pm delete program Subroutine" on page 818), pm get data ("pm get data, pm get tdata, pm get data cpu, and pm get tdata cpu Subroutine" on page 824), pm start ("pm start Subroutine" on page 851), pm stop ("pm_stop Subroutine" on page 856), pm_reset_data ("pm_reset_data Subroutine" on page 839) subroutines.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_get_program_group Subroutine

Purpose

Retrieves the Performance Monitor settings for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_get_program_group ( pid, tid, *prog)
pid t pid;
tid_t tid;
pm_prog_t *prog;
```

Description

The pm_get_program_group subroutine retrieves the Performance Monitor settings for the counting group to which a target kernel thread belongs. The thread must be stopped and must be part of a debuggee process under the control of the calling process. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the pm init subroutine.

The counting mode includes the user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a threshold value is also returned.

Parameters

pid Process identifier of target thread. The target process must be an argument of a debug process. tid Thread identifier of the target thread.

*prog

Returns which Performance Monitor events and modes are set. Supported modes are:

PM_USER

Counting process running in user mode

PM KERNEL

Counting process running kernel mode

PM COUNT

Counting is on

PM PROCESS

Process level counting group

Return Values

No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm set program group ("pm set program group Subroutine" on page 845) subroutine, pm delete program group ("pm delete program group Subroutine" on page 819) subroutine, pm_get_data_group ("pm_get_data_group and pm_get_tdata_group Subroutine" on page 825) subroutine, pm start group ("pm start group Subroutine" on page 852) subroutine, pm stop group ("pm stop group Subroutine" on page 857) subroutine, pm_reset_data_group ("pm_reset_data_group Subroutine" on page 840) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_get_program_mygroup Subroutine

Purpose

Retrieves the Performance Monitor settings for the counting group to which the calling thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
```

```
int pm_get_program_mygroup ( *prog)
pm_prog_t *prog;
```

Description

The pm_get_program_mygroup subroutine retrieves the Performance Monitor settings for the counting group to which the calling kernel thread belongs. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the pm_init subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value is also returned.

Parameters

*prog

Returns which Performance Monitor events and modes are set. Supported modes are:

PM_USER

Counting processes running in user mode

PM_KERNEL

Counting processes running in kernel mode

PM_COUNT

Counting is on

PM_PROCESS

Process level counting group

Return Values

No errors occurred.

Positive error code Refer to the pm error ("pm error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_set_program_mygroup ("pm_set_program_mygroup Subroutine" on page 847) subroutine, pm_delete_program_mygroup ("pm_delete_program_mygroup Subroutine" on page 820) subroutine, pm_get_data_mygroup ("pm_get_data_mygroup or pm_get_tdata_mygroup Subroutine" on page 827) subroutine, pm_start_mygroup ("pm_start_mygroup Subroutine" on page 853) subroutine, pm_stop_mygroup ("pm_stop_mygroup Subroutine" on page 858) subroutine, pm_reset_data_mygroup ("pm_reset_data_mygroup Subroutine" on page 841) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_get_program_mythread Subroutine

Purpose

Retrieves the Performance Monitor settings for the calling thread.

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm get program mythread ( *prog)
pm_prog_t *prog;
```

Description

The pm_get_program_mythread subroutine retrieves the Performance Monitor settings for the calling kernel thread. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the pm init subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a threshold value is also returned.

Parameters

Returns which Performance Monitor events and modes *prog

are set. Supported modes are:

PM_USER

Counting processes running in user mode

PM KERNEL

Counting processes running in kernel mode

PM COUNT

Counting is on

Return Values

No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_set_program_mythread ("pm_set_program_mythread Subroutine" on page 848) subroutine, pm_delete_program_mythread ("pm_delete_program_mythread Subroutine" on page 821) subroutine, pm_get_data_mythread ("pm_get_data_mythread or pm_get_tdata_mythread Subroutine" on page 828) subroutine, pm start mythread ("pm start mythread Subroutine" on page 854) subroutine, pm_stop_mythread ("pm_stop_mythread Subroutine" on page 859) subroutine, pm_reset_data_mythread ("pm_reset_data_mythread Subroutine" on page 842) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm get program thread Subroutine

Purpose

Retrieves the Performance Monitor settings for a target thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm get program thread ( pid, tid, *prog)
pid t pid;
tid t tid;
pm_prog_t *prog;
```

Description

The pm get program thread subroutine retrieves the Performance Monitor settings for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of the calling process. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the **pm** init subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the pm init subroutine), a threshold value is also returned.

Parameters

pid tid *prog Process identifier of the target thread. The target process must be an argument of a debug process.

Thread identifier of the target thread.

Returns which Performance Monitor events and modes are set. Supported modes are:

PM_USER

Counting processes running in User mode

PM KERNEL

Counting processes running in Kernel mode

PM COUNT

Counting is On

Return Values

No errors occurred.

Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

pm init ("pm init Subroutine"), pm error ("pm_error Subroutine" on page 823), pm_set_program_thread ("pm_set_program_thread Subroutine" on page 850), pm_delete_program_thread ("pm_delete_program_thread Subroutine" on page 822), pm_get_data_thread ("pm_get_data_thread or pm_get_tdata_thread Subroutine" on page 829), pm_start_thread ("pm_start_thread Subroutine" on page 855), pm_stop_thread ("pm_stop_thread Subroutine" on page 860), pm_reset_data_thread ("pm_reset_data_thread Subroutine" on page 843) subroutines.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_init Subroutine

Purpose

Initializes the Performance Monitor APIs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_init ( filter, *pminfo, *pm_groups_info)
int filter;
pm_info_t *pminfo;
pm_groups_info_t *pm_groups_info;
```

Description

The pm_init subroutine initializes the Performance Monitor API library. It returns, after taking into account a filter on the status of the events, the number of counters available on this processor, and one table per counter with the list of events available. For each event, an event identifier, a status, a flag indicating if the event can be used with a threshold, two names, and a description are provided.

The event identifier is used with all the pm_set_program interfaces and is also returned by all of the pm get program interfaces. Only event identifiers present in the table returned can be used. In other words, the filter is effective for all API calls.

The status describes whether the event has been verified, is still unverified, or works with some caveat, as explained in the description. This field is necessary because the filter can be any combination of the three available status bits. The flag points to events that can be used with a threshold.

Only events categorized as verified have gone through full verification. Events categorized as caveat have been verified only within the limitations documented in the event description. Events categorized as unverified have undefined accuracy. Use caution with unverified events; the Performance Monitor software is essentially providing a service to read hardware registers which may or may not have any meaningful content. Users may experiment with unverified event counters and determine for themselves what, if any, use they may have for specific tuning situations.

The short mnemonic name is provided for easy keyword-based search in the event table (see the sample program /usr/samples/pmapi/sysapit2.c for code using mnemonic names). The complete name of the event is also available and a full description for each event is returned.

The structure returned also has the threshold multiplier for this processor and the processor name

On some platforms, it is possible to specify event groups instead of individual events. Event groups are predefined sets of events. Rather than specify each event individually, a single group ID is specified. On some platforms, such as POWER4, use of the event groups is required, and attempts to specify individual events return an error.

The interface to **pm** init has been enhanced to return the list of supported event groups in an optional third parameter. For binary compatibility, the third parameter must be explicitly requested by OR-ing the bitflag, PM GET GROUPS, into the filter parameter.

If the pm groups info parameter returned by pm init is NULL, there are no supported event groups for the platform. Otherwise an array of pm groups t structures are returned in the event groups field. The length of the array is given by the max groups field.

The pm_groups_t structure contains a group identifier, two names and a description that are similar to those of the individual events. In addition, there is an array of integers that specify the events contained in the group.

Parameters

filter Specifies which event types to return.

> **PM VERIFIED** Events which have been verified

PM UNVERIFIED

Events which have not been verified

PM_CAVEAT

Events which are usable but with caveats as described in the long description

Returned structure with processor name, threshold multiplier, and a filtered list of events with their current status.

Returned structure with list of supported groups. This parameter is only meaningful if PM_GET_GROUPS is OR-ed into the *filter* parameter.

*pm_groups_info

*pminfo

Return Values

0 No errors occurred. Positive error code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

pm_error ("pm_error Subroutine" on page 823) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_reset_data Subroutine

Purpose

Resets system wide Performance Monitor data.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_reset_data ()

Description

The pm reset data subroutine resets the current system wide Performance Monitor data. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0.

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

See the pm_error ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The **pm_init** ("pm_init Subroutine" on page 837) subroutine, **pm_error** ("pm_error Subroutine" on page 823) subroutine, **pm_set_program** ("pm_set_program Subroutine" on page 844) subroutine, **pm_get_program** ("pm_get_program Subroutine" on page 831) subroutine, **pm_delete program** ("pm_delete_program Subroutine" on page 818) subroutine, **pm_get_data** ("pm_get_data, pm_get_tdata, pm_get_data_cpu, and pm_get_tdata_cpu Subroutine" on page 824) subroutine, **pm_start** ("pm_start Subroutine" on page 851) subroutine, **pm_stop** ("pm_stop Subroutine" on page 856) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_reset_data_group Subroutine

Purpose

Resets Performance Monitor data for a target thread and the counting group to which it belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_reset_data_group ( pid, tid)
pid_t pid;
tid_t tid;
```

Description

The pm_reset_data_group subroutine resets the current Performance Monitor data for a target kernel thread and the counting group to which it belongs. The thread must be stopped and must be part of a debugee process, under control of the calling process. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0. Because the data for all the other threads in the group is not affected, the group is marked as inconsistent unless it has only one member.

Parameters

pid
Process ID of target thread. Target process must be a debuggee of the caller process.

tid
Thread ID of target thread.

Return Values

Operation completed successfully.

Positive Error Code Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm init ("pm init Subroutine" on page 837) subroutine, pm error ("pm error Subroutine" on page 823) subroutine, pm set program group ("pm set program group Subroutine" on page 845) subroutine, pm_get_program_group ("pm_get_program_group Subroutine" on page 832) subroutine, pm delete program group ("pm delete program group Subroutine" on page 819) subroutine, pm start group ("pm start group Subroutine" on page 852) subroutine, pm stop group ("pm stop group Subroutine" on page 857) subroutine, pm get data group ("pm get data group and pm get tdata group Subroutine" on page 825) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_reset_data_mygroup Subroutine

Purpose

Resets Performance Monitor data for the calling thread and the counting group to which it belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_reset_data_mygroup()

Description

The pm_reset_data_mygroup subroutine resets the current Performance Monitor data for the calling kernel thread and the counting group to which it belongs. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0. Because the data for all the other threads in the group is not affected, the group is marked as inconsistent unless it has only one member.

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm init ("pm init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_set_program_mygroup ("pm_set_program_mygroup Subroutine" on page 847) subroutine, pm_get_program_mygroup ("pm_get_program_mygroup Subroutine" on page 833) subroutine, pm_delete_program_mygroup ("pm_delete_program_mygroup Subroutine" on page 820) subroutine, pm_start_mygroup ("pm_start_mygroup Subroutine" on page 853) subroutine, pm_stop_mygroup ("pm_stop_mygroup Subroutine" on page 858) subroutine, pm_get_data_mygroup ("pm_get_data_mygroup or pm_get_tdata_mygroup Subroutine" on page 827) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm reset data mythread Subroutine

Purpose

Resets Performance Monitor data for the calling thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_reset_data_mythread()

Description

The pm reset data mythread subroutine resets the current Performance Monitor data for the calling kernel thread. The data is a set (one per hardware counter on the machine) of 64-bit values. All values are reset to 0.

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm error** ("pm error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The **pm_init** ("pm_init Subroutine" on page 837) subroutine, **pm_error** ("pm_error Subroutine" on page 823) subroutine, pm_set_program_mythread ("pm_set_program_mythread Subroutine" on page 848) subroutine, pm_get_program_mythread ("pm_get_program_mythread Subroutine" on page 835) subroutine, pm_delete_program_mythread ("pm_delete_program_mythread Subroutine" on page 821) subroutine, pm start mythread ("pm start mythread Subroutine" on page 854) subroutine,

pm_stop_mythread ("pm_stop_mythread Subroutine" on page 859) subroutine, pm_get_data_mythread ("pm get data mythread or pm get tdata mythread Subroutine" on page 828) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm reset_data_thread Subroutine

Purpose

Resets Performance Monitor data for a target thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_reset_data_thread ( pid, tid)
pid t pid;
tid_t tid;
```

Description

The pm reset data thread subroutine resets the current Performance Monitor data for a target kernel thread. The thread must be stopped and must be part of a debuggee process. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0.

Parameters

Process id of target thread. Target process must be a pid debuggee of the caller process.

Thread id of target thread. tid

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, datatypes, and subroutines.

Related Information

The pm init ("pm init Subroutine" on page 837) subroutine, pm error ("pm error Subroutine" on page 823) subroutine, pm set program thread ("pm set program thread Subroutine" on page 850) subroutine, pm_get_program_thread ("pm_get_program_thread Subroutine" on page 836) subroutine, pm delete program thread ("pm delete program thread Subroutine" on page 822) subroutine,

pm_start_thread ("pm_start_thread Subroutine" on page 855) subroutine, pm_stop_thread ("pm stop thread Subroutine" on page 860) subroutine, pm get data thread ("pm get data thread or pm_get_tdata_thread Subroutine" on page 829) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_set_program Subroutine

Purpose

Sets system wide Performance Monitor programmation.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h> int pm set program (*prog) pm_prog_t *prog;

Description

The pm set program subroutine sets system wide Performance Monitor programmation. The setting includes the events to be counted, and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the pm init subroutine.

The counting mode includes User Mode and/or Kernel Mode, the Initial Counting State, and the Process Tree Mode. The Process Tree Mode sets counting to On only for the calling process and its descendants. The defaults are set to Off for User Mode and Kernel Mode. The initial default state is set to delay counting until the pm start subroutine is called, and to count the activity of all the processes running in the system.

If the list includes an event which can be used with a threshold (as indicated by the pm_init subroutine), a threshold value can also be specified.

On some platforms, event groups can be specified instead of individual events. This is done by setting the bitfield is group in the mode, and placing the group ID into the first element of the events array. (The group ID was obtained by pm init).

Parameters

*prog

Specifies the events and modes to use in Performance Monitor setup. The following modes are supported:

PM USER

Counts processes running in User Mode (default is set to Off)

PM_KERNEL

Counts processes running in Kernel Mode (default is set to Off)

PM_COUNT

Starts counting immediately (default is set to Not to Start Counting)

PM PROCTREE

Sets counting to On only for the calling process and its descendants (default is set to Off)

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_get_program ("pm_get_program Subroutine" on page 831) subroutine, pm delete program ("pm delete program Subroutine" on page 818) subroutine, pm get data ("pm_get_data, pm_get_tdata, pm_get_data_cpu, and pm_get_tdata_cpu Subroutine" on page 824) subroutine, pm_start ("pm_start Subroutine" on page 851) subroutine, pm_stop ("pm_stop Subroutine" on page 856) subroutine, pm_reset_data ("pm_reset_data Subroutine" on page 839) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_set_program_group Subroutine

Purpose

Sets Performance Monitor programmation for a target thread and creates a counting group.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm set program group ( pid, tid, *prog)
pid t pid;
tid t tid;
pm prog t *prog;
```

Description

The pm_set_program_group subroutine sets the Performance Monitor programmation for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the pm_init subroutine.

This call also creates a counting group, which includes the target thread and any thread which it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future threads belonging to the target process.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the pm start group subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the pm init subroutine), a threshold value can also be specified.

Parameters

pid

tid *prog Process ID of target thread. Target process must be a debuggee of a calling process.

Thread ID of target thread.

PM USER

Counts processes running in User Mode (default is set to Off)

PM KERNEL

Counts processes running in Kernel Mode (default is set to Off)

PM COUNT

Starts counting immediately (default is set to Not to Start Counting)

PM PROCESS

Creates a process-level counting group

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h

Defines standard macros, data types, and subroutines.

Related Information

The pm init ("pm init Subroutine" on page 837) subroutine, pm error ("pm error Subroutine" on page 823) subroutine, pm get program group ("pm get program group Subroutine" on page 832) subroutine, pm_delete_program_group ("pm_delete_program_group Subroutine" on page 819) subroutine, pm get data group ("pm get data group and pm get tdata group Subroutine" on page 825) subroutine, pm start group ("pm start group Subroutine" on page 852) subroutine, pm stop group ("pm_stop_group Subroutine" on page 857) subroutine, pm_reset data group ("pm reset data group Subroutine" on page 840) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_set_program_mygroup Subroutine

Purpose

Sets Performance Monitor programmation for the calling thread and creates a counting group.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm set program mygroup (*prog)
pm_prog_t *prog;
```

Description

The pm_set_program_mygroup subroutine sets the Performance Monitor programmation for the calling kernel thread. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the pm init subroutine.

This call also creates a counting group, which includes the calling thread and any thread which it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future threads belonging to the calling process.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the inital default state is set to delay counting until the pm_start_mygroup subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the pm init subroutine), a threshold value can also be specified.

Parameters

*prog

Specifies the events and mode to use in Performance Monitor setup. The following modes are supported:

Counts processes running in User Mode (default is set to Off)

PM_KERNEL

Counts processes running in Kernel Mode (default is set to Off)

PM_COUNT

Starts counting immediately (default is set to Not to Start Counting)

PM PROCESS

Creates a process-level counting group

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h

Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm get program mygroup ("pm get program mygroup Subroutine" on page 833) subroutine, pm delete program mygroup ("pm delete program mygroup Subroutine" on page 820) subroutine, pm get data mygroup ("pm get data mygroup or pm get tdata mygroup Subroutine" on page 827) subroutine, pm_start_mygroup ("pm_start_mygroup Subroutine" on page 853) subroutine, pm_stop_mygroup ("pm_stop_mygroup Subroutine" on page 858) subroutine, pm_reset_data_mygroup ("pm_reset_data_mygroup Subroutine" on page 841) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_set_program_mythread Subroutine

Purpose

Sets Performance Monitor programmation for the calling thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm set program mythread (*prog) pm prog t *prog;

Description

The pm_set_program_mythread subroutine sets the Performance Monitor programmation for the calling kernel thread. The setting includes the events to be counted, and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the pm init subroutine.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the pm_start_mythread subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the pm init subroutine), a threshold value can also be specified.

Parameters

*prog

Specifies the event modes to use in Performance Monitor setup. The following modes are supported:

PM USER

Counts processes running in User Mode (default is set to Off)

PM_KERNEL

Counts processes running in Kernel Mode (default is set to Off)

PM COUNT

Starts counting immediately (default is set to Not to Start Counting)

PM PROCESS

Creates a process-level counting group

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm error** ("pm error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm init ("pm init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_get_program_mythread ("pm_get_program_mythread Subroutine" on page 835) subroutine, pm_delete_program_mythread ("pm_delete_program_mythread Subroutine" on page 821) subroutine, pm_get_data_mythread ("pm_get_data_mythread or pm_get_tdata_mythread Subroutine" on page 828) subroutine, pm start mythread ("pm start mythread Subroutine" on page 854) subroutine, pm_stop_mythread ("pm_stop_mythread Subroutine" on page 859) subroutine, pm_reset_data_mythread ("pm_reset_data_mythread Subroutine" on page 842) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm set program thread Subroutine

Purpose

Sets Performance Monitor programmation for a target thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm set program thread ( pid, tid, *prog)
pid t pid;
tid t tid;
pm_prog_t *prog;
```

Description

The pm set program thread subroutine sets the Performance Monitor programmation for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the pm init subroutine.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the Initial Default State is set to delay counting until the pm_start_thread subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the pm init subroutine), a threshold value can also be specified.

Parameters

Process ID of target thread. Target process must be a pid debuggee of the caller process. tid Thread ID of target thread.

*prog

Specifies the event modes to use in Performance Monitor setup. The following modes are supported:

PM_USER

Counts processes running in User Mode (default is set to Off)

PM_KERNEL

Counts processes running in Kernel Mode (default is set to Off)

PM_COUNT

Starts counting immediately (default is set to Not to Start Counting)

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_get_program_thread ("pm_get_program_thread Subroutine" on page 836) subroutine, pm_delete_program_thread ("pm_delete_program_thread Subroutine" on page 822) subroutine, pm_get_data_thread ("pm_get_data_thread or pm_get_tdata_thread Subroutine" on page 829) subroutine, pm_start_thread ("pm_start_thread Subroutine" on page 855) subroutine, pm_stop_thread ("pm_stop_thread Subroutine" on page 860) subroutine, pm_reset_data_thread ("pm_reset_data_thread Subroutine" on page 843) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_start Subroutine

Purpose

Starts system wide Performance Monitor counting.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm start()

Description

The pm start subroutine starts system wide Performance Monitor counting.

Return Values

Operation completed successfully.

Positive Error Code. Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code

Error Codes

Refer to the **pm error** ("pm error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm init ("pm init Subroutine" on page 837) subroutine, pm error ("pm error Subroutine" on page 823) subroutine, pm_set_program ("pm_set_program Subroutine" on page 844) subroutine, pm_get_program ("pm_get_program Subroutine" on page 831) subroutine, pm_delete_program ("pm delete program Subroutine" on page 818) subroutine, pm get data ("pm get data, pm get tdata, pm get data cpu, and pm get tdata cpu Subroutine" on page 824) subroutine, pm stop ("pm stop Subroutine" on page 856) subroutine, **pm reset data** ("pm reset data Subroutine" on page 839) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_start_group Subroutine

Purpose

Starts Performance Monitor counting for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm start group (pid, tid)
pid_t pid;
tid t tid;
```

Description

The pm start group subroutine starts the Performance Monitor counting for a target kernel thread and the counting group to which it belongs. This counting is effective immediately for the target thread. For all the other thread members of the counting group, the counting will start after their next redispatch, but only if their current counting state is already set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state. If their counting state is currently set to Off, no

counting starts until they call either the pm start mythread subroutine or the pm start mygroup themselves, or until a debugger process calls the pm start thread subroutine or the pm start group subroutine on their behalf.

Parameters

pid Process ID of target thread. Target process must be a

> debuggee of the caller process. Thread ID of target thread.

tid

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm error** ("pm error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm set program group ("pm set program group Subroutine" on page 845) subroutine, pm_get_program_group ("pm_get_program_group Subroutine" on page 832) subroutine, pm_delete_program_group ("pm_delete_program_group Subroutine" on page 819) subroutine. pm get data group ("pm get data group and pm get tdata group Subroutine" on page 825) subroutine, pm_stop_group ("pm_stop_group Subroutine" on page 857) subroutine, pm_reset_data_group ("pm_reset_data_group Subroutine" on page 840) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_start_mygroup Subroutine

Purpose

Starts Performance Monitor counting for the group to which the calling thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int_pm_start_mygroup()

Description

The pm_start_mygroup subroutine starts the Performance Monitor counting for the calling kernel thread and the counting group to which it belongs. Counting is effective immediately for the calling thread. For all the other threads members of the counting group, the counting starts after their next redispatch, but only if their current counting state is already set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state. If their counting state is currently set to Off, no counting starts until they call either the pm_start_mythread subroutine or the pm_start_mygroup subroutine themselves, or until a debugger process calls the pm_start_thread subroutine or the pm_start_group subroutine on their behalf.

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_set_program_mygroup ("pm_set_program_mygroup Subroutine" on page 847) subroutine, pm_get_program_mygroup ("pm_get_program_mygroup Subroutine" on page 833) subroutine, pm_delete_program_mygroup ("pm_delete_program_mygroup Subroutine" on page 820) subroutine, pm_get_data_mygroup ("pm_get_data_mygroup or pm_get_tdata_mygroup Subroutine" on page 827) subroutine, pm_stop_mygroup ("pm_stop_mygroup Subroutine" on page 858) subroutine, pm_reset_data_mygroup ("pm_reset_data_mygroup Subroutine" on page 841) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_start_mythread Subroutine

Purpose

Starts Performance Monitor counting for the calling thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm start mythread()

Description

The pm start mythread subroutine starts Performance Monitor counting for the calling kernel thread. Counting is effective immediately unless the thread is in a group, and that group's counting is not currently set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state.

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm init ("pm init Subroutine" on page 837) subroutine, pm error ("pm error Subroutine" on page 823) subroutine, pm set program mythread ("pm set program mythread Subroutine" on page 848) subroutine, pm get program mythread ("pm get program mythread Subroutine" on page 835) subroutine, pm_delete_program_mythread ("pm_delete_program_mythread Subroutine" on page 821) subroutine, pm get data mythread ("pm get data mythread or pm get tdata mythread Subroutine" on page 828) subroutine, pm_stop_mythread ("pm_stop_mythread Subroutine" on page 859) subroutine, pm reset data mythread ("pm reset data mythread Subroutine" on page 842) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm start thread Subroutine

Purpose

Starts Performance Monitor counting for a target thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_start_thread ( pid, tid)
pid t pid;
tid t tid;
```

Description

The pm_start_thread subroutine starts Performance Monitor counting for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process. Counting is effective immediately unless the thread is in a group and the group counting is not currently set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state.

Parameters

pid Process ID of target thread. Target process must be a

debuggee of the caller process.

tid Thread ID of target thread.

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm error** ("pm error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm set program thread ("pm set program thread Subroutine" on page 850) subroutine, pm_get_program_thread ("pm_get_program_thread Subroutine" on page 836) subroutine, pm delete program thread ("pm delete program thread Subroutine" on page 822) subroutine, pm get data thread ("pm get data thread or pm get tdata thread Subroutine" on page 829) subroutine, pm stop thread ("pm stop thread Subroutine" on page 860) subroutine, pm reset data thread ("pm reset data thread Subroutine" on page 843) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_stop Subroutine

Purpose

Stops system wide Performance Monitor counting.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_stop ()

Description

The **pm_stop** subroutine stops system wide Performance Monitoring counting.

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm error** ("pm error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm set program ("pm set program Subroutine" on page 844) subroutine, pm get program ("pm get program Subroutine" on page 831) subroutine, pm delete program ("pm delete program Subroutine" on page 818) subroutine, pm get data ("pm get data, pm get tdata, pm_get_data_cpu, and pm_get_tdata_cpu Subroutine" on page 824) subroutine, pm_start ("pm_start Subroutine" on page 851) subroutine, pm reset data ("pm reset data Subroutine" on page 839) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_stop_group Subroutine

Purpose

Stops Performance Monitor counting for the group to which a target thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_stop_group ( pid, tid)
pid_t pid;
tid_t tid;
```

Description

The **pm** stop subroutine stops Performance Monitor counting for a target kernel thread, the counting group to which it belongs, and all the other thread members of the same group. Counting stops immediately for all the threads in the counting group. The target thread must be stopped and must be part of a debuggee process, under control of the calling process.

Parameters

pid

Process ID of target thread. Target process must be a debuggee of the caller process.

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm_init ("pm_init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_set_program_group ("pm_set_program_group Subroutine" on page 845) subroutine, pm_get_program_group ("pm_get_program_group Subroutine" on page 832) subroutine, pm_delete_program_group ("pm_delete_program_group Subroutine" on page 819) subroutine, pm_get_data_group ("pm_get_data_group and pm_get_tdata_group Subroutine" on page 825) subroutine, pm_start_group ("Syntax" on page 852) subroutine, pm_reset_data_group ("pm_reset_data_group Subroutine" on page 840) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_stop_mygroup Subroutine

Purpose

Stops Performance Monitor counting for the group to which the calling thread belongs.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_stop_mygroup ()

Description

The **pm_stop_mygroup** subroutine stops Performance Monitor counting for the group to which the calling kernel thread belongs. This is effective immediately for all the threads in the counting group.

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm init ("pm init Subroutine" on page 837) subroutine, pm error ("pm error Subroutine" on page 823) subroutine, pm set program mygroup ("pm set program mygroup Subroutine" on page 847) subroutine, pm_get_program_mygroup ("pm_get_program_mygroup Subroutine" on page 833) subroutine, pm delete program mygroup ("pm delete program mygroup Subroutine" on page 820) subroutine, pm_get_data_mygroup ("pm_get_data_mygroup or pm_get_tdata_mygroup Subroutine" on page 827) subroutine, pm_start_mygroup ("Description" on page 854) subroutine, pm reset data mygroup ("pm reset data mygroup Subroutine" on page 841) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_stop_mythread Subroutine

Purpose

Stops Performance Monitor counting for the calling thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

#include <pmapi.h>

int pm_stop_mythread ()

Description

The pm_stop_mythread subroutine stops Performance Monitor counting for the calling kernel thread.

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm error ("pm error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The **pm_init** ("pm_init Subroutine" on page 837) subroutine, **pm_error** ("pm_error Subroutine" on page 823) subroutine, **pm_set_program_mythread** ("pm_set_program_mythread Subroutine" on page 848) subroutine, **pm_get_program_mythread** ("pm_get_program_mythread Subroutine" on page 835) subroutine, **pm_delete_program_mythread** ("pm_delete_program_mythread Subroutine" on page 821) subroutine, **pm_get_data_mythread** ("pm_get_data_mythread or pm_get_tdata_mythread Subroutine" on page 828) subroutine, **pm_start_mythread** ("pm_start_mythread Subroutine" on page 854) subroutine, **pm_reset_data_mythread** ("pm_reset_data_mythread Subroutine" on page 842) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

pm_stop_thread Subroutine

Purpose

Stops Performance Monitor counting for a target thread.

Library

Performance Monitor APIs Library (libpmapi.a)

Syntax

```
#include <pmapi.h>
int pm_stop_thread ( pid, tid)
pid_t pid;
tid t tid;
```

Description

The **pm_stop_thread** subroutine stops Performance Monitor counting for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process.

Parameters

pid Process ID of target thread. Target process must be a debuggee of the caller process.

tid Thread ID of target thread.

Return Values

Operation completed successfully.

Positive Error Code Refer to the pm_error ("pm_error Subroutine" on page 823) subroutine to decode the

error code.

Error Codes

Refer to the **pm_error** ("pm_error Subroutine" on page 823) subroutine.

Files

/usr/include/pmapi.h Defines standard macros, data types, and subroutines.

Related Information

The pm init ("pm init Subroutine" on page 837) subroutine, pm_error ("pm_error Subroutine" on page 823) subroutine, pm_set_program_thread ("pm_set_program_thread Subroutine" on page 850) subroutine, pm_get_program_thread ("pm_get_program_thread Subroutine" on page 836) subroutine, pm_delete_program_thread ("pm_delete_program_thread Subroutine" on page 822) subroutine, pm_get_data_thread ("pm_get_data_thread or pm_get_tdata_thread Subroutine" on page 829) subroutine, pm_start_thread ("pm_start_thread Subroutine" on page 855) subroutine, pm_reset_data_thread ("pm_reset_data_thread Subroutine" on page 843) subroutine.

Performance Monitor API Programming Concepts in AIX 5L Version 5.2 Performance Tools Guide and Reference.

poll Subroutine

Purpose

Checks the I/O status of multiple file descriptors and message queues.

Library

Standard C Library (libc.a)

Syntax

```
#include <sys/poll.h>
#include <sys/select.h>
#include <sys/types.h>
int poll( ListPointer, Nfdsmsgs, Timeout)
void *ListPointer;
unsigned long Nfdsmsgs;
long Timeout;
```

Description

The **poll** subroutine checks the specified file descriptors and message queues to see if they are ready for reading (receiving) or writing (sending), or to see if they have an exceptional condition pending. Even though there are OPEN MAX number of file descriptors available, only FD SETSIZE number of file descriptors are accessible with this subroutine.

Note: The poll subroutine applies only to character devices, pipes, message queues, and sockets. Not all character device drivers support it. See the descriptions of individual character devices for information about whether and how specific device drivers support the **poll** and **select** subroutines.

For compatibility with previous releases of this operating system and with BSD systems, the select subroutine is also supported.

Parameters

ListPointer

Specifies a pointer to an array of pollfd structures, pollmsg structures, or to apollist structure. Each structure specifies a file descriptor or message queue ID and the events of interest for this file or message queue. The pollfd, pollmsg, and pollist structures are defined in the /usr/include/sys/poll.h file. If a pollist structure is to be used, a structure similar to the following should be defined in a user program. The pollfd structure must precede the pollmsg structure.

```
struct pollist {
  struct pollfd fds[3];
  struct pollmsg msgs[2];
  } list;
```

The structure can then be initialized as follows:

```
list.fds[0].fd = file descriptorA;
list.fds[0].events = requested_events;
list.msgs[0].msgid = message_id;
list.msgs[0].events = requested events;
```

The rest of the elements in thefdsandmsgsarrays can be initialized the same way. The poll subroutine can then be called, as follows:

```
nfds = 3;
          /* number of pollfd structs */
nmsgs = 2;
           /* number of pollmsg structs */
timeout = 1000 /* number of milliseconds to timeout */
poll(&list, (nmsgs<<16) | (nfds), 1000);
```

The exact number of elements in the fds and msgs arrays must be used in the calculation of the Nfdsmsgs parameter.

Nfdsmsgs

Specifies the number of file descriptors and the exact number of message queues to check. The low-order 16 bits give the number of elements in the array of pollfd structures, while the high-order 16 bits give the exact number of elements in the array of pollmsg structures. If either half of the Nfdsmsgs parameter is equal to a value of 0, the corresponding array is assumed not to be present.

Timeout

Specifies the maximum length of time (in milliseconds) to wait for at least one of the specified events to occur. If the Timeout parameter value is -1, the poll subroutine does not return until at least one of the specified events has occurred. If the value of the Timeout parameter is 0, the poll subroutine does not wait for an event to occur but returns immediately, even if none of the specified events have occurred.

poll Subroutine STREAMS Extensions

In addition to the functions described above, the poll subroutine multiplexes input/output over a set of file descriptors that reference open streams. The poll subroutine identifies those streams on which you can send or receive messages, or on which certain events occurred.

You can receive messages using the **read** subroutine or the **getmsg** system call. You can send messages using the write subroutine or the putmsq system call. Certain streamio operations, such as I RECVFD and I SENDFD can also be used to send and receive messages. See the streamio operations.

The ListPointer parameter specifies the file descriptors to be examined and the events of interest for each file descriptor. It points to an array having one element for each open file descriptor of interest. The array's elements are pollfd structures. In addition to the pollfd structure in the /usr/include/sys/poll.h file, STREAMS supports the following members:

```
int fd;
                  /* file descriptor */
                  /* requested events */
short events;
                  /* returned events */
short revents;
```

The fd field specifies an open file descriptor and the events and revents fields are bit-masks constructed by ORing any combination of the following event flags:

POLLIN A nonpriority or file descriptor-passing message is present on the stream-head read

queue. This flag is set even if the message is of 0 length. In the revents field this flag is

mutually exclusive with the **POLLPRI** flag. See the **I_RECVFD** command.

POLLRDNORM A nonpriority message is present on the stream-head read queue.

A priority message (band > 0) is present on the stream-head read queue. POLLRDBAND

POLLPRI A high-priority message is present on the stream-head read queue. This flag is set even if

the message is of 0 length. In the revents field, this flag is mutually exclusive with the

POLLIN flag.

POLLOUT The first downstream write queue in the stream is not full. Normal priority messages can

be sent at any time. See the putmsg system call.

The same as POLLOUT. **POLLWRNORM**

POLLWRBAND A priority band greater than 0 exists downstream and priority messages can be sent at

POLLMSG A message containing the SIGPOLL signal has reached the front of the stream-head read

queue.

Return Values

On successful completion, the poll subroutine returns a value that indicates the total number of file descriptors and message queues that satisfy the selection criteria. The return value is similar to the Nfdsmsgs parameter in that the low-order 16 bits give the number of file descriptors, and the high-order 16 bits give the number of message queue identifiers that had nonzero revents values. The NFDS and NMSGS macros, found in the sys/select.h file, can be used to separate these two values from the return value. The NFDS macro returns NFDS#, where the number returned indicates the number of files reporting some event or error, and the NMSGS macro returns NMSGS#, where the number returned indicates the number of message queues reporting some event or error.

A value of 0 indicates that the **poll** subroutine timed out and that none of the specified files or message queues indicated the presence of an event (all revents fields were values of 0).

If unsuccessful, a value of -1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The poll subroutine does not run successfully if one or more of the following are true:

EAGAIN Allocation of internal data structures was unsuccessful.

EINTR A signal was caught during the poll system call and the signal handler was installed with an indication

that subroutines are not to be restarted.

EINVAL The number of **pollfd** structures specified by the *Nfdsmsgs* parameter is greater than **FD_SETSIZE**. This

error is also returned if the number of pollmsg structures specified by the Nfdsmsgs parameter is greater

than the maximum number of allowable message queues.

EFAULT The ListPointer parameter in conjunction with the Nfdsmsgs parameter addresses a location outside of

the allocated address space of the process.

Related Information

The **read** subroutine, **select** subroutine, **write** subroutine.

The **getmsg** system call, **putmsg** system call.

The **streamio** operations.

The STREAMS Overview and the Input and Output Handling Programmer's Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

popen Subroutine

Purpose

Initiates a pipe to a process.

Library

Standard C Library (libc.a)

Syntax

#include <stdio.h>

FILE *popen (Command, Type)
const char *Command, *Type;

Description

The **popen** subroutine creates a pipe between the calling program and a shell command to be executed.

Note: The **popen** subroutine runs only **sh** shell commands. The results are unpredictable if the *Command* parameter is not a valid **sh** shell command. If the terminal is in a trusted state, the **tsh** shell commands are run.

If streams opened by previous calls to the **popen** subroutine remain open in the parent process, the **popen** subroutine closes them in the child process.

The **popen** subroutine returns a pointer to a **FILE** structure for the stream.

Attention: If the original processes and the process started with the **popen** subroutine concurrently read or write a common file, neither should use buffered I/O. If they do, the results are unpredictable.

Some problems with an output filter can be prevented by flushing the buffer with the **fflush** subroutine.

Parameters

Command Type Points to a null-terminated string containing a shell command line.

Points to a null-terminated string containing an I/O mode. If the Type parameter is the value \mathbf{r} , you can read from the standard output of the command by reading from the file Stream. If the Type parameter is the value \mathbf{w} , you can write to the standard input of the command by writing to the file Stream.

Because open files are shared, a type ${\bf r}$ command can be used as an input filter and a type ${\bf w}$ command as an output filter.

Return Values

The **popen** subroutine returns a null pointer if files or processes cannot be created, or if the shell cannot be accessed.

Error Codes

The **popen** subroutine may set the **EINVAL** variable if the *Type* parameter is not valid. The **popen** subroutine may also set **errno** global variables as described by the **fork** or **pipe** subroutines.

Related Information

The fclose or fflush ("fclose or fflush Subroutine" on page 213) subroutine, fopen, freopen, or fdopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, fork or vfork ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, pclose ("pclose Subroutine" on page 797) subroutine, pipe ("pipe Subroutine" on page 815) subroutine, wait, waitpid, or wait3 subroutine.

Input and Output Handling.

File Systems and Directories in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

posix openpt Subroutine

Purpose

Opens a pseudo-terminal device.

Library

Standard C library (libc.a)

Syntax

```
#include <stdlib.h<</pre>
#include <fcntl.h>
int posix_openpt (oflag
int oflag;
```

Description

The **posix openpt** subroutine establishes a connection between a master device for a pseudo terminal and a file descriptor. The file descriptor is used by other I/O functions that refer to that pseudo terminal.

The file status flags and file access modes of the open file description are set according to the value of the oflag parameter.

Parameters

oflag

Values for the oflag parameter are constructed by a bitwise-inclusive OR of flags from the following list, defined in the <fcntl.h> file:

O RDWR

Open for reading and writing.

O NOCTTY

If set, the posix_openpt subroutine does not cause the terminal device to become the controlling terminal for the process.

The behavior of other values for the oflag parameter is unspecified.

Return Values

Upon successful completion, the posix_openpt subroutine opens a master pseudo-terminal device and returns a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, -1 is returned and the errno global variable is set to indicate the error.

Error Codes

The **posix_openpt** subroutine will fail if:

EMFILE OPEN_MAX file descriptors are currently open in the calling process. **ENFILE** The maximum allowable number of files is currently open in the system.

The posix_openpt subroutine may fail if:

EINVAL The value of the oflag parameter is not valid.

EAGAIN Out of pseudo-terminal resources. **ENOSR** Out of STREAMS resources.

Examples

The following example describes how to open a pseudo-terminal and return the name of the slave device and file descriptor

```
#include <fcntl.h>
#include <stdio.h>
int masterfd, slavefd;
char *slavedevice;
masterfd = posix openpt(0 RDWR|0 NOCTTY);
if (masterfd == -1)
   | | grantpt (masterfd) == -1
      unlockpt (masterfd) == -1
      (slavedevice = ptsname (masterfd)) == NULL)
     return -1;
printf("slave device is: %s\n", slavedevice);
slavefd = open(slave, 0 RDWR|0 NOCTTY);
if (slavefd < 0)
  return -1;
```

Related Information

"grantpt Subroutine" on page 404, "open, openx, open64, creat, or creat64 Subroutine" on page 763, "ptsname Subroutine" on page 989.

unlockpt Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

<fcntl.h> file in AIX 5L Version 5.2 Files Reference.

powf, powl, or pow Subroutine

Purpose

Computes power.

Syntax

```
#include <math.h>
float powf (x, y)
float x;
float y;
```

```
long double powl (x, y)
long double x, y;
double pow (x, y)
double x, y;
```

Description

The **powf**, **powl**, and **pow** subroutines compute the value of x raised to the power y, x^y . If x is negative, the application ensures that y is an integer value.

An application wishing to check for error situations should set errno to zero and call feclearexcept(FE_ALL_EXCEPT) before calling these subroutines. Upon return, if errno is nonzero or fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is nonzero, an error has occurred.

Parameters

Specifies the value of the base.

Specifies the value of the exponent.

Return Values

Upon successful completion, the **pow**, **powf** and **powl** subroutines return the value of x raised to the power y.

For finite values of x < 0, and finite non-integer values of y, a domain error occurs and a NaN is returned.

If the correct value would cause overflow, a range error occurs and the pow, powf, and powl subroutines return HUGE VAL, HUGE VALF, and HUGE VALL, respectively.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 is returned.

If x or y is a NaN, a NaN is returned (unless specified elsewhere in this description).

For any value of y (including NaN), if x is +1, 1.0 is returned.

For any value of x (including NaN), if y is ± 0 , 1.0 is returned.

For any odd integer value of y>0, if x is ± 0 , ± 0 is returned.

For y > 0 and not an odd integer, if x is ± 0 , ± 0 is returned.

If x is -1, and y is $\pm \ln f$, 1.0 is returned.

For |x<1, if y is $-\ln f$, $+\ln f$ is returned.

For |x>1, if y is $-\ln f$, +0 is returned.

For |x<1, if y is +Inf, +0 is returned.

For |x>1, if y is +Inf, +Inf is returned.

For y an odd integer < 0, if x is -Inf, -0 is returned.

For y < 0 and not an odd integer, if x is -lnf, +0 is returned.

For y an odd integer > 0, if x is $-\ln f$, $-\ln f$ is returned.

For y > 0 and not an odd integer, if x is -lnf, +lnf is returned.

For y < 0, if x is $+ \ln f$, + 0 is returned.

For y > 0, if x is $+ \ln f$, $+ \ln f$ is returned.

For y an odd integer < 0, if x is ± 0 , a pole error occurs and $\pm HUGE_VAL$, $\pm HUGE_VALF$, and $\pm HUGE_VALL$ is returned for **pow**, **powf**, and **powl**, respectively.

For y < 0 and not an odd integer, if x is ± 0 , a pole error occurs and **HUGE_VAL**, **HUGE_VALF** and **HUGE_VALL** is returned for **pow**, **powf**, and **powl**, respectively.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

Error Codes

When using the libm.a library:

pow

If the correct value overflows, the **pow**subroutine returns a **HUGE_VAL** value and sets **errno** to **ERANGE**. If the x parameter is negative and the y parameter is not an integer, the **pow** subroutine returns a **NaNQ** value and sets **errno** to **EDOM**. If x=0 and the y parameter is negative, the **pow** subroutine returns a **HUGE_VAL** value but does not modify **errno**.

powl

If the correct value overflows, the **powl**subroutine returns a **HUGE_VAL** value and sets **errno** to **ERANGE**. If the x parameter is negative and the y parameter is not an integer, the **powl** subroutine returns a **NaNQ** value and sets **errno** to **EDOM**. If x=0 and the y parameter is negative, the **powl** subroutine returns a **HUGE_VAL** value but does not modify **errno**.

When using libmsaa.a(-lmsaa):

pow

If x=0 and the y parameter is not positive, or if the x parameter is negative and the y parameter is not an integer, the **pow** subroutine returns 0 and sets **errno** to **EDOM**. In these cases a message indicating DOMAIN error is output to standard error. When the correct value for the **pow** subroutine would overflow or underflow, the **pow** subroutine returns:

```
+HUGE VAL
```

OR

-HUGE VAL

0R

0

When using either the libm.a library or the libsaa.a library:

powl

If the correct value overflows, **powl** returns **HUGE_VAL** and **errno** to **ERANGE**. If x is negative and y is not an integer, **powl** returns **NaNQ** and sets **errno** to **EDOM**. If x = z ero and y is negative, **powl** returns a **HUGE_VAL** value but does not modify **errno**.

Related Information

"exp, expf, or expl Subroutine" on page 205, "feclearexcept Subroutine" on page 223, "fetestexcept Subroutine" on page 231, and "class, _class, finite, isnan, or unordered Subroutines" on page 138.

printf, fprintf, sprintf, snprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine

Purpose

Prints formatted output.

Library

Standard C Library (libc.a) or the Standard C Library with 128-Bit long doubles (libc128.a)

Syntax

```
#include <stdio.h>
int printf (Format, [Value, ...])
const char *Format;
int fprintf (Stream, Format, [Value, ...])
FILE *Stream;
const char *Format;
int sprintf (String, Format, [Value, ...])
char *String;
const char *Format;
int snprintf (String, Number, Format, [Value, . . .])
char *String;
int Number;
const char *Format;
#include <stdarg.h>
int vprintf (Format, Value)
const char *Format;
va_list Value;
int vfprintf (Stream, Format, Value)
FILE *Stream;
const char *Format;
va list Value;
int vsprintf (String, Format, Value)
char *String;
const char *Format;
va list Value;
#include <wchar.h>
int vwsprintf (String, Format, Value)
wchar t *String;
const char *Format;
va_list Value;
int wsprintf (String, Format, [Value, ...])
wchar_t *String;
const char *Format;
```

Description

The **printf** subroutine converts, formats, and writes the *Value* parameter values, under control of the *Format* parameter, to the standard output stream. The **printf** subroutine provides conversion types to handle code points and **wchar_t** wide character codes.

The fprintf subroutine converts, formats, and writes the Value parameter values, under control of the Format parameter, to the output stream specified by the Stream parameter. This subroutine provides conversion types to handle code points and wchar_t wide character codes.

The sprintf subroutine converts, formats, and stores the Value parameter values, under control of the Format parameter, into consecutive bytes, starting at the address specified by the String parameter. The sprintf subroutine places a null character (\0) at the end. You must ensure that enough storage space is available to contain the formatted string. This subroutine provides conversion types to handle code points and wchar_t wide character codes.

The **snprintf** subroutine converts, formats, and stores the *Value* parameter values, under control of the Format parameter, into consecutive bytes, starting at the address specified by the String parameter. The snprintf subroutine places a null character (\0) at the end. You must ensure that enough storage space is available to contain the formatted string. This subroutine provides conversion types to handle code points and wchar_t wide character codes. The snprintf subroutine is identical to the sprintf subroutine with the addition of the *Number* parameter, which states the size of the buffer referred to by the *String* parameter.

The wsprintf subroutine converts, formats, and stores the Value parameter values, under control of the Format parameter, into consecutive wchar t characters starting at the address specified by the String parameter. The wsprintf subroutine places a null character (\0) at the end. The calling process should ensure that enough storage space is available to contain the formatted string. The field width unit is specified as the number of wchar t characters. The wsprintf subroutine is the same as the printf subroutine, except that the String parameter for the wsprintf subroutine uses a string of wchar t wide-character codes.

All of the above subroutines work by calling the _doprnt subroutine, using variable-length argument facilities of the varargs macros.

The vprintf, vfprintf, vsprintf, and vwsprintf subroutines format and write varargs macros parameter lists. These subroutines are the same as the printf, fprintf, sprintf, snprintf, and wsprintf subroutines, respectively, except that they are not called with a variable number of parameters. Instead, they are called with a parameter-list pointer as defined by the varargs macros.

Parameters

Number

Specifies the number of bytes in a string to be copied or transformed.

Value Specifies 0 or more arguments that map directly to the objects in the Format parameter.

Stream

Specifies the output stream.

String Specifies the starting address.

Format

A character string that contains two types of objects:

- · Plain characters, which are copied to the output stream.
- Conversion specifications, each of which causes 0 or more items to be retrieved from the Value parameter list. In the case of the vprintf, vfprintf, vsprintf, and vwsprintf subroutines, each conversion specification causes 0 or more items to be retrieved from the varargs macros parameter lists.

If the Value parameter list does not contain enough items for the Format parameter, the results are unpredictable. If more parameters remain after the entire Format parameter has been processed, the subroutine ignores them.

Each conversion specification in the *Format* parameter has the following elements:

A % (percent sign).

- 0 or more options, which modify the meaning of the conversion specification. The option characters and their meanings are:
 - Formats the integer portions resulting from i, d, u, f, g and G decimal conversions with thousands sep grouping characters. For other conversions the behavior is undefined. This option uses the nonmonetary grouping character.
 - Left-justifies the result of the conversion within the field.
 - Begins the result of a signed conversion with a + (plus sign) or (minus sign).

space character

Prefixes a space character to the result if the first character of a signed conversion is not a sign. If both the space-character and + option characters appear, the space-character option is ignored.

- # Converts the value to an alternate form. For c, d, s, and u conversions, the option has no effect. For o conversion, it increases the precision to force the first digit of the result to be a 0. For x and X conversions, a nonzero result has a 0x or 0X prefix. For e, E, f, a, and G conversions, the result always contains a decimal point, even if no digits follow it. For **g** and **G** conversions, trailing 0's are not removed from the result.
- 0 Pads to the field width with leading 0's (following any indication of sign or base) for d, i, o, u, x, X, e, E, f, q, and G conversions; the field is not space-padded. If the 0 and options both appear, the 0 option is ignored. For d, i, o u, x, and X conversions, if a precision is specified, the **0** option is also ignored. If the **0** and 'options both appear, grouping characters are inserted before the field is padded. For other conversions, the results are unreliable.
- В Specifies a no-op character.
- Ν Specifies a no-op character.
- J Specifies a no-op character.
- · An optional decimal digit string that specifies the minimum field width. If the converted value has fewer characters than the field width, the field is padded on the left to the length specified by the field width. If the - (left-justify) option is specified, the field is padded on the right.
- An optional precision. The precision is a . (dot) followed by a decimal digit string. If no precision is specified, the default value is 0. The precision specifies the following limits:
 - Minimum number of digits to appear for the d, i, o, u, x, or X conversions.
 - Number of digits to appear after the decimal point for the **e**, **E**, and **f** conversions.
 - Maximum number of significant digits for g and G conversions.
 - Maximum number of bytes to be printed from a string in s and S conversions.
 - Maximum number of bytes, converted from the wchar t array, to be printed from the S conversions. Only complete characters are printed.
- An optional I (lowercase L), II (lowercase LL), h, or L specifier indicates one of the following:
 - An optional **h** specifying that a subsequent **d**, **i**, **u**, **o**, **x**, or **X** conversion specifier applies to a short int or unsigned short int Value parameter (the parameter will have been promoted according to the integral promotions, and its value will be converted to a short int or unsigned short int before printing).
 - An optional h specifying that a subsequent n conversion specifier applies to a pointer to a **short int** parameter.
 - An optional I (lowercase L) specifying that a subsequent d, i, u, o, x, or X conversion specifier applies to a long int or unsigned long int parameter.
 - An optional I (lowercase L) specifying that a subsequent n conversion specifier applies to a pointer to a long int parameter.

- An optional II (lowercase LL) specifying that a subsequent d, i, u, o, x, or X conversion specifier applies to a long long int or unsigned long long int parameter.
- An optional II (lowercase LL) specifying that a subsequent n conversion specifier applies to a pointer to a **long long int** parameter.
- An optional L specifying that a following e, E, f, g, or G conversion specifier applies to a long double parameter. If linked with libc.a, long double is the same as double (64bits). If linked with libc128.a and libc.a, long double is 128 bits.
- The following characters indicate the type of conversion to be applied:
 - % Performs no conversion. Prints (%).
 - d or i Accepts a Value parameter specifying an integer and converts it to signed decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field-width value to be padded with leading 0's.
 - Accepts a Value parameter specifying an unsigned integer and converts it to unsigned u decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field-width value to be padded with leading 0's.
 - Accepts a Value parameter specifying an unsigned integer and converts it to unsigned 0 octal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field-width with a 0 as a leading character causes the field width value to be padded with leading 0's. An octal value for field width is not implied.
 - x or X Accepts a Value parameter specifying an unsigned integer and converts it to unsigned hexadecimal notation. The letters abcdef are used for the x conversion and the letters ABCDEF are used for the X conversion. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field-width value to be padded with leading 0's.
 - f Accepts a Value parameter specifying a double and converts it to decimal notation in the format [-]ddd.ddd. The number of digits after the decimal point is equal to the precision specification. If no precision is specified, six digits are output. If the precision is 0, no decimal point appears.
 - e or E Accepts a Value parameter specifying a double and converts it to the exponential form [-]d.ddde+/-dd. One digit exists before the decimal point, and the number of digits after the decimal point is equal to the precision specification. The precision specification can be in the range of 0-17 digits. If no precision is specified, six digits are output. If the precision is 0, no decimal point appears. The E conversion character produces a number with E instead of e before the exponent. The exponent always contains at least two digits.
 - g or G

Accepts a Value parameter specifying a double and converts it in the style of the e, E, or f conversion characters, with the precision specifying the number of significant digits. Trailing 0's are removed from the result. A decimal point appears only if it is followed by a digit. The style used depends on the value converted. Style e (E, if G is the flag used)

- results only if the exponent resulting from the conversion is less than -4, or if it is greater or equal to the precision. If an explicit precision is 0, it is taken as 1.
- С Accepts and prints a Value parameter specifying an integer converted to an unsigned char data type.
- C Accepts and prints a Value parameter specifying a wchar t wide character code. The wchar_t wide character code specified by the Value parameter is converted to an array of bytes representing a character and that character is written; the Value parameter is written without conversion when using the wsprintf subroutine.
- S Accepts a Value parameter as a string (character pointer), and characters from the string are printed until a null character (\0) is encountered or the number of bytes indicated by the precision is reached. If no precision is specified, all bytes up to the first null character are printed. If the string pointer specified by the Value parameter has a null value, the results are unreliable.
- S Accepts a corresponding Value parameter as a pointer to a wchar t string. Characters from the string are printed (without conversion) until a null character (\0) is encountered or the number of wide characters indicated by the precision is reached. If no precision is specified, all characters up to the first null character are printed. If the string pointer specified by the Value parameter has a value of null, the results are unreliable.
- Accepts a pointer to void. The value of the pointer is converted to a sequence of р printable characters, the same as an unsigned hexadecimal (x).
- Accepts a pointer to an integer into which is written the number of characters n (wide-character codes in the case of the wsprintf subroutine) written to the output stream by this call. No argument is converted.

A field width or precision can be indicated by an * (asterisk) instead of a digit string. In this case, an integer Value parameter supplies the field width or precision. The Value parameter converted for output is not retrieved until the conversion letter is reached, so the parameters specifying field width or precision must appear before the value (if any) to be converted.

If the result of a conversion is wider than the field width, the field is expanded to contain the converted result and no truncation occurs. However, a small field width or precision can cause truncation on the right.

The printf, sprintf, sprintf, sprintf, wsprintf, vprintf, vprintf, vsprintf, or vwsprintf subroutine allows the insertion of a language-dependent radix character in the output string. The radix character is defined by language-specific data in the LC NUMERIC category of the program's locale. In the C locale, or in a locale where the radix character is not defined, the radix character defaults to a . (dot).

After any of these subroutines runs successfully, and before the next successful completion of a call to the fclose ("fclose or fflush Subroutine" on page 213) or fflush subroutine on the same stream or to the exit ("exit, atexit, exit, or Exit Subroutine" on page 203) or abort ("abort Subroutine" on page 3) subroutine, the st ctime and st mtime fields of the file are marked for update.

The e, E, f, q, and G conversion specifiers represent the special floating-point values as follows:

Quiet NaN +NaNQ or -NaNQ Signaling NaN +NaNS or -NaNS +/-INF +INF or -INF +/-0 +0 or -0

The representation of the + (plus sign) depends on whether the + or space-character formatting option is specified.

These subroutines can handle a format string that enables the system to process elements of the parameter list in variable order. In such a case, the normal conversion character % (percent sign) is replaced by %digit\$, where digit is a decimal number in the range from 1 to the NL_ARGMAX value. Conversion is then applied to the specified argument, rather than to the next unused argument. This feature provides for the definition of format strings in an order appropriate to specific languages. When variable ordering is used the * (asterisk) specification for field width in precision is replaced by %digit\$. If you use the variable-ordering feature, you must specify it for all conversions.

The following criteria apply:

- The format passed to the NLS extensions can contain either the format of the conversion or the explicit or implicit argument number. However, these forms cannot be mixed within a single format string, except for %% (double percent sign).
- The n value must have no leading zeros.
- If %n\$ is used, %1\$ to %n 1\$ inclusive must be used.
- The n in %n\$ is in the range from 1 to the **NL ARGMAX** value, inclusive. See the **limits.h** file for more information about the NL ARGMAX value.
- · Numbered arguments in the argument list can be referenced as many times as required.
- The * (asterisk) specification for field width or precision is not permitted with the variable order %n\$ format; instead, the *m\$ format is used.

Return Values

Upon successful completion, the printf, fprintf, vprintf, and vfprintf subroutines return the number of bytes transmitted (not including the null character [\0] in the case of the **sprintf**, and **vsprintf** subroutines). If an error was encountered, a negative value is output.

Upon successful completion, the **snprintf** subroutine returns the number of bytes written to the *String* parameter (excluding the terminating null byte). If output characters are discarded because the output exceeded the Number parameter in length, then the snprintf subroutine returns the number of bytes that would have been written to the String parameter if the Number parameter had been large enough (excluding the terminating null byte).

Upon successful completion, the wsprintf and vwsprintf subroutines return the number of wide characters transmitted (not including the wide character null character [\0]). If an error was encountered, a negative value is output.

Error Codes

The printf, sprintf, sprintf, or wsprintf subroutine is unsuccessful if the file specified by the Stream parameter is unbuffered or the buffer needs to be flushed and one or more of the following are true:

EAGAIN	The O_NONBLOCK flag is set for the file descriptor underlying the file specified by the <i>Stream</i> or <i>String</i>
	parameter and the process would be delayed in the write operation.

EBADF	The file descriptor underlying the file specified by the Stream or String parameter is not a valid file
	descriptor open for writing.

EFBIG An attempt was made to write to a file that exceeds the file size limit of this process or the maximum file size. For more information, refer to the **ulimit** subroutine.

EINTR The write operation terminated due to receipt of a signal, and either no data was transferred or a partial transfer was not reported.

Note: Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding **sa** restart.

EIO The process is a member of a background process group attempting to perform a write to its controlling

terminal, the TOSTOP flag is set, the process is neither ignoring nor blocking the SIGTTOU signal, and

the process group of the process has no parent process.

ENOSPC No free space remains on the device that contains the file.

EPIPE An attempt was made to write to a pipe or first-in-first-out (FIFO) that is not open for reading by any

process. A SIGPIPE signal is sent to the process.

The printf, sprintf, sprintf, or wsprintf subroutine may be unsuccessful if one or more of the following are true:

EILSEQ An invalid character sequence was detected.

EINVAL The Format parameter received insufficient arguments.

ENOMEM Insufficient storage space is available.

ENXIO A request was made of a nonexistent device, or the request was outside the capabilities of the device.

Examples

The following example demonstrates how the **vfprintf** subroutine can be used to write an error routine:

```
#include <stdio.h>
#include <stdarg.h>
/* The error routine should be called with the
     syntax:
                   */
/* error(routine name, Format
     [, value, . . . ]); */
/*VARARGS0*/
void error(char *fmt, . . .);
/* ** Note that the function name and
    Format arguments cannot be **
     separately declared because of the **
     definition of varargs. */ {
   va_list args;
   va start(args, fmt);
   /*
   ** Display the name of the function
     that called the error routine */
   fprintf(stderr, "ERROR in %s: ",
     va arg(args, char *));
   ** Display the remainder of the message
   fmt = va arg(args, char *);
   vfprintf(fmt, args);
   va end(args);
   abort(); }
```

Related Information

The abort ("abort Subroutine" on page 3) subroutine, conv ("conv Subroutines" on page 149) subroutine, ecvt, fcvt, or gcvt ("ecvt, fcvt, or gcvt Subroutine" on page 185) subroutine, exit ("exit, atexit, exit, or _Exit Subroutine" on page 203) subroutine, fclose or fflush ("fclose or fflush Subroutine" on page 213) subroutine, putc, putchar, fputc, or putw ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine, putwc, putwchar, or fputwc ("putwc, putwchar, or fputwc Subroutine" on page 995) subroutine, scanf, fscanf, sscanf, or wsscanf subroutine, setlocale subroutine.

Input and Output Handling and 128-Bit Long Double Floating-Point Data Type in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

profil Subroutine

Purpose

Starts and stops program address sampling for execution profiling.

Library

Standard C Library (libc.a)

Syntax

#include <mon.h>

void profil (ShortBuffer, BufferSize, Offset, Scale) OR void profil (ProfBuffer, -1, 0, 0)

unsigned short *ShortBuffer; struct prof *ProfBuffer; unsigned int Buffersize. Scale: unsigned long Offset;

Description

The profil subroutine arranges to record a histogram of periodically sampled values of the calling process program counter. If BufferSize is not -1:

- The parameters to the **profil** subroutine are interpreted as shown in the first syntax definition.
- · After this call, the program counter (pc) of the process is examined each clock tick if the process is the currently active process. The value of the Offset parameter is subtracted from the pc. The result is multiplied by the value of the Scale parameter, shifted right 16 bits, and rounded up to the next half-word aligned value. If the resulting number is less than the BufferSize value divided by sizeof(short), the corresponding short inside the ShortBuffer parameter is incremented. If the result of this increment would overflow an unsigned short, it remains USHRT_MAX.
- The least significant 16 bits of the Scale parameter are interpreted as an unsigned, fixed-point fraction with a binary point at the left. The most significant 16 bits of the Scale parameter are ignored. For example:

Octal	Hex	Meaning
0177777	0xFFFF	Maps approximately each pair of bytes in the instruction space to a unique short in the <i>ShortBuffer</i> parameter.
077777	0x7FFF	Maps approximately every four bytes to a short in the <i>ShortBuffer</i> parameter.
02	0x0002	Maps all instructions to the same location, producing a noninterrupting core clock.
01	0x0001	Turns profiling off.
00	0x0000	Turns profiling off.

Note: Mapping each byte of the instruction space to an individualshort in the ShortBuffer parameter is not possible.

 Profiling, using the first syntax definition, is rendered ineffective by giving a value of 0 for the BufferSize parameter.

If the value of the *BufferSize* parameter is -1:

 The parameters to the profil subroutine are interpreted as shown in the second syntax definition. In this case, the Offset and Scale parameters are ignored, and the ProfBuffer parameter points to an array of prof structures. The prof structure is defined in the mon.h file, and it contains the following members:

p_low; caddr t caddr t p high; HISTCOUNTER *p_buff; int p_bufsize; uint p scale;

If the p scale member has the value of -1, a value for it is computed based on p low, p high, and p bufsize; otherwise p scale is interpreted like the scale argument in the first synopsis. The p high members in successive structures must be in ascending sequence. The array of structures is ended with a structure containing a p high member set to 0; all other fields in this last structure are ignored.

The p buff buffer pointers in the array of **prof** structures must point into a single contiguous buffer space.

 Profiling, using the second syntax definition, is turned off by giving a ProfBuffer argument such that the p high element of the first structure is equal to 0.

In every case:

- Profiling remains on in both the child process and the parent process after a fork subroutine.
- Profiling is turned off when an exec subroutine is run.
- A call to the profil subroutine is ineffective if profiling has been previously turned on using one syntax definition, and an attempt is made to turn profiling off using the other syntax definition.
- A call to the profil subroutine is ineffective if the call is attempting to turn on profiling when profiling is already turned on, or if the call is attempting to turn off profiling when profiling is already turned off.

Parameters

ShortBuffer Points to an area of memory in the user address space. Its length (in bytes) is given by the

BufferSize parameter.

BufferSize Specifies the length (in bytes) of the buffer.

Offset Specifies the delta of program counter start and buffer; for example, a 0 Offset implies that text

begins at 0. If the user wants to use the entry point of a routine for the Offset parameter, the

syntax of the parameter is as follows:

*(long *)RoutineName

Scale Specifies the mapping factor between the program counter and ShortBuffer.

ProfBuffer Points to an array of **prof** structures.

Return Values

The profil subroutine always returns a value of 0. Otherwise, the errno global variable is set to indicate the error.

Error Codes

The **profil** subroutine is unsuccessful if one or both of the following are true:

EFAULT The address specified by the ShortBuffer or ProfBuffer parameters is not valid, or the address specified

by a p buff field is not valid. EFAULT can also occur if there are not sufficient resources to pin the

profiling buffer in real storage.

The p high fields in the prof structure specified by the ProfBuffer parameter are not in ascending order. **EINVAL**

Related Information

The **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutines, fork ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, moncontrol ("moncontrol") Subroutine" on page 696) subroutine, monitor ("monitor Subroutine" on page 697) subroutine, monstartup ("monstartup Subroutine" on page 703) subroutine.

The **prof** command.

psdanger Subroutine

Purpose

Defines the amount of free paging space available.

Syntax

#include <signal.h> #include <sys/vminfo.h>

blkcnt_t psdanger (Signal) int Signal;

Description

The **psdanger** subroutine returns the difference between the current number of free paging-space blocks and the paging-space thresholds of the system.

Parameters

Signal Defines the signal.

Return Values

If the value of the Signal parameter is 0, the return value is the total number of paging-space blocks defined in the system.

If the value of the Signal parameter is -1, the return value is the number of free paging-space blocks available in the system.

If the value of the Signal parameter is SIGDANGER, the return value is the difference between the current number of free paging-space blocks and the paging-space warning threshold. If the number of free paging-space blocks is less than the paging-space warning threshold, the return value is negative.

If the value of the Signal parameter is SIGKILL, the return value is the difference between the current number of free paging-space blocks and the paging-space kill threshold. If the number of free paging-space blocks is less than the paging-space kill threshold, the return value is negative.

Related Information

The **swapoff** subroutine, **swapon** subroutine, **swapqry** subroutine.

The chps command, Isps command, mkps command, rmps command, swapoff command, swapon command.

Paging Space Overview in AIX 5L Version 5.2 System Management Concepts: Operating System and Devices.

Subroutines Overview and Understanding Paging Space Programming Requirements in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

psignal Subroutine or sys_siglist Vector

Purpose

Prints system signal messages.

Library

Standard C Library (libc.a)

Syntax

psignal (Signal, String) unsigned Signal; char *String; char *sys siglist[];

Description

The psignal subroutine produces a short message on the standard error file describing the indicated signal. First the String parameter is printed, then the name of the signal and a new-line character.

To simplify variant formatting of signal names, the sys_siglist vector of message strings is provided. The signal number can be used as an index in this table to get the signal name without the new-line character. The NSIG defined in the signal.h file is the number of messages provided for in the table. It should be checked because new signals may be added to the system before they are added to the table.

Parameters

Signal Specifies a signal. The signal number should be among those found in the signal.h file.

Specifies a string that is printed. Most usefully, the String parameter is the name of the program that String incurred the signal.

Related Information

The **perror** ("perror Subroutine" on page 814) subroutine, **sigvec** subroutine.

pthdb_attr, pthdb_cond, pthdb_condattr, pthdb_key, pthdb_mutex, pthdb_mutexattr, pthdb_pthread, pthdb_pthread_key, pthdb_rwlock, or pthdb_rwlockattr Subroutine

Purpose

Reports the pthread library objects.

Library

pthread debug library (libpthdebug.a)

Syntax

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

int pthdb_pthread (pthdb_session_t session,

```
pthdb pthread t * pthreadp,
                                    cmd)
int pthdb_pthread_key(pthdb_session_t session,
                      pthread_key_t * keyp,
                                       cmd)
int pthdb attr(pthdb session t session,
               pthdb attr t
                               * attrp,
               int
                                cmd)
int pthdb_cond (pthdb_session_t session,
                pthdb cond t
                                * condp,
                int
                                 cmd)
int pthdb_condattr (pthdb_session t
                                      session,
                    pthdb condattr t * condattrp,
                                      cmd)
int pthdb key(pthdb session t session,
              pthdb pthread t pthread,
              pthread key t * keyp,
              int
                               cmd)
int pthdb mutex (pthdb session t session,
                 pthdb mutex t
                                 * mutexp,
int pthdb mutexattr (pthdb session t
                                        session,
                     pthdb mutexattr t * mutexattrp,
                     int
int pthdb rwlock (pthdb session t session,
                  pthdb rwlock t * rwlockp,
                  int
                                   cmd)
int pthdb rwlockattr (pthdb session t
                      pthdb rwlockattr t * rwlockattrp,
                      int
                                          cmd)
```

Description

The pthread library maintains internal lists of objects: pthreads, mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, read/write lock attributes, attributes, pthread specific keys, and active keys. The pthread debug library provides access to these lists one element at a time via the functions listed above.

Each one of those functions acquire the next element in the list of objects. For example, the **pthdb_attr** function gets the next attribute on the list of attributes.

A report of PTHDB_INVALID_OBJECT represents the empty list or the end of a list, where OBJECT is equal to PTHREAD, ATTR, MUTEX, MUTEXATTR, COND, CONDATTR, RWLOCK, RWLOCKATTR, KEY, or TID as appropriate.

Each list is reset to the top of the list when the **pthdb_session_update** function is called, or when the list function reports a **PTHDB_INVALID_*** value. For example, when **pthdb_attr** reports an attribute of **PTHDB_INVALID_ATTR** the list is reset to the beginning such that the next call reports the first attribute in the list, if any.

When **PTHDB LIST FIRST** is passed for the *cmd* parameter, the first item in the list is retrieved.

Parameters

session Session handle.

attrp Attribute object.

cmd Reset to the beginning of the list.condp Pointer to Condition variable object.

condattrp Pointer to Condition variable attribute object.

keyp Pointer to Key object.

mutexattrp Pointer to Mutex attribute object.

mutexp Pointer to Mutex object.

pthread pthread object.

pthreadp Pointer to pthread object.

rwlockp Pointer to Read/Write lock object.

rwlockattrp Pointer to Read/Write lock attribute object.

Return Values

If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB_BAD_SESSION Invalid session handle.
PTHDB_BAD_PTHREAD Invalid pthread handle.
PTHDB_BAD_CMD Invalid command.
PTHDB_BAD_POINTER Invalid buffer pointer.
PTHDB_INTERNAL Error in library.
PTHDB_MEMORY Not enough memory

Related Information

The pthdebug.h file.

The pthread.h file.

```
pthdb_attr_detachstate,pthdb_attr_addr,
pthdb_attr_guardsize,pthdb_attr_inheritsched,
pthdb_attr_schedparam,pthdb_attr_schedpolicy,
pthdb_attr_schedpriority,pthdb_attr_scope,
pthdb_attr_stackaddr,pthdb_attr_stacksize, or pthdb_attr_suspendstate
Subroutine
```

Purpose

Query the various fields of a pthread attribute and return the results in the specified buffer.

Library

pthread debug library (libpthdebug.a)

Syntax

```
int pthdb attr guardsize (pthdb session t
                                              session.
                          pthdb attr t
                          pthdb size t * guardsizep);
int pthdb attr inheritsched (pthdb session t
                                                   session,
                             pthdb attr t
                                                   attr,
                             pthdb inheritsched t * inheritschedp);
int pthdb attr schedparam (pthdb session t
                                               session,
                           pthdb attr t
                                               attr,
                           struct sched param * schedparamp);
int pthdb_attr_schedpolicy (pthdb_session_t session,
                            pthdb attr t
                                            attr,
                            pthdb policy t * schedpolicyp)
int pthdb_attr_schedpriority (pthdb_session t session,
                              pthdb attr t
                                              * schedpriorityp)
int pthdb attr scope (pthdb session t session,
                      pthdb attr t
                      pthdb scope t * scopep)
int pthdb attr stackaddr (pthdb session t
                                               session.
                          pthdb attr t
                                               attr.
                          pthdb_size_t * stackaddrp);
int pthdb attr stacksize (pthdb session t
                                               session,
                          pthdb attr t
                                              attr.
                          pthdb_size_t * stacksizep);
int pthdb_attr_suspendstate (pthdb_session t
                                                   session,
                             pthdb_attr_t
                                                  attr,
                             pthdb suspendstate t * suspendstatep)
```

Description

Each pthread is created using either the default pthread attribute or a user-specified pthread attribute. These functions query the various fields of a pthread attribute and, if successful, return the result in the buffer specified. In all cases, the values returned reflect the expected fields of a pthread created with the attribute specified.

pthdb_attr_detachstate reports if the created pthread is detachable (**PDS_DETACHED**) or joinable (**PDS_JOINABLE**). **PDS_NOTSUP** is reserved for unexpected results.

pthdb attr addr reports the address of the pthread attr t.

pthdb attr **guardsize** reports the guard size for the attribute.

pthdb_attr_inheritsched reports whether the created pthread will run with scheduling policy and scheduling parameters from the created pthread (PIS_INHERIT), or from the attribute (PIS_EXPLICIT).
PIS_NOTSUP is reserved for unexpected results.

pthdb_attr_schedparam reports the scheduling parameters associated with the pthread attribute. See **pthdb attr inheritsched** for additional information.

pthdb_attr_schedpolicy reports whether the scheduling policy associated with the pthread attribute is other (**SP_OTHER**), first in first out (**SP_FIFO**), or round robin (**SP_RR**). **SP_NOTSUP** is reserved for unexpected results.

pthdb_attr_schedpriority reports the scheduling priority associated with the pthread attribute. See **pthdb_attr_inheritsched** for additional information.

pthdb_attr_scope reports whether the created pthread will have process scope (**PS_PROCESS**) or system scope (**PS_SYSTEM**). **PS_NOTSUP** is reserved for unexpected results.

pthdb_attr_stackaddr reports the address of the stack.

pthdb_attr_stacksize reports the size of the stack.

pthdb_attr_suspendstate reports whether the created pthread will be suspended (**PSS_SUSPENDED**) or not (**PSS_UNSUSPENDED**). **PSS_NOTSUP** is reserved for unexpected results.

Parameters

addr Attributes address. attr Attributes handle. detachstatep Detach state buffer. guardsizep Attribute guard size. Inherit scheduling buffer. inheritschedp schedparamp Scheduling parameters buffer. schedpolicyp Scheduling policy buffer. schedpriorityp Scheduling priority buffer. scopep Contention scope buffer.

session Session handle.

stackaddrpAttributes stack address.stacksizepAttributes stack size.suspendstatepSuspend state buffer.

Return Values

If successful these functions return PTHDB_SUCCESS. Otherwise, and error code is returned.

Error Codes

PTHDB_BAD_SESSION Invalid session handle.
PTHDB_BAD_ATTR Invalid attribute handle.
PTHDB_BAD_POINTER Invalid buffer pointer.
PTHDB_CALLBACK Debugger call back error.

PTHDB_NOTSUP Not supported.
PTHDB_INTERNAL Internal library error.

Related Information

The pthdebug.h file.

The pthread.h file.

pthdb_condattr_pshared, or pthdb_condattr_addr Subroutine

Purpose

Gets the condition variable attribute pshared value.

Library

pthread debug library (libpthdebug.a)

Syntax

#include <sys/pthdebug.h>

Description

The **pthdb_condattr_pshared** function is used to get the condition variable attribute process shared value. The pshared value can be **PSH_SHARED**, **PSH_PRIVATE**, or **PSH_NOTSUP**.

The pthdb_condattr_addr function reports the address of the pthread_condattr_t.

Parameters

addrp Pointer to the address of the pthread_condattr_t.

condattrCondition variable attribute handlepsharedpPointer to the pshared value.

session Session handle.

Return Values

If successful this function returns PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB_BAD_CONDATTR Invalid condition variable attribute handle.

PTHDB_BAD_SESSION Invalid session handle.
PTHDB_CALLBACK Debugger call back error.

PTHDB_INTERNAL Error in library.
PTHDB_POINTER Invalid pointer

Related Information

The pthdebug.h file.

The pthread.h file.

pthdb_cond_addr, pthdb_cond_mutex or pthdb_cond_pshared Subroutine

Purpose

Gets the condition variable's mutex handle and pshared value.

Library

pthread debug library (libpthdebug.a)

Syntax

#include <sys/pthdebug.h>

Description

The pthdb_cond_addr function reports the address of the pthdb_cond_t.

The **pthdb_cond_mutex** function is used to get the mutex handle associated with the particular condition variable, if the mutex does not exist then PTHDB_INVALID_MUTEX is returned from the mutex.

The **pthdb_cond_pshared** function is used to get the condition variable process shared value. The pshared value can be **PSH_SHARED**, **PSH_PRIVATE**, or **PSH_NOTSUP**.

Parameters

addrCondition variable addresscondCondition variable handlemutexpPointer to mutexpsharedpPointer to pshared valuesessionSession handle.

Return Values

If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB_BAD_COND Invalid cond handle.
PTHDB_BAD_SESSION Invalid session handle.
PTHDB_CALLBACK Debugger call back error.
PTHDB_INVALID_MUTEX Invalid mutex.
PTHDB_INTERNAL Error in library.
PTHDB_POINTER Invalid pointer

Related Information

The pthdebug.h file.

The pthread.h file.

pthdb_mutexattr_addr, pthdb_mutexattr_prioceiling, pthdb_mutexattr_protocol, pthdb_mutexattr_pshared or pthdb_mutexattr_type Subroutine

Purpose

Gets the mutex attribute pshared, priority ceiling, protocol, and type values.

Library

pthread debug library (libpthdebug.a)

Syntax

Description

The pthdb_mutexattr_addr function reports the address of the pthread_mutexatt_t.

The pthdb_mutexattr_prioceiling function is used to get the mutex attribute priority ceiling value.

The **pthdb_mutexattr_protocol** function is used to get the mutex attribute protocol value. The protocol value can be **MP INHERIT**, **MP PROTECT**, **MP NONE**, or **MP NOTSUP**.

The **pthdb_mutexattr_pshared** function is used to get the mutex attribute process shared value. The pshared value can be **PSH_SHARED**, **PSH_PRIVATE**, or **PSH_NOTSUP**.

The **pthdb_mutexattr_type** is used to get the value of the mutex attribute type. The values for the mutex type can be **MK_NONRECURSIVE_NP**, **MK_RECURSIVE_NP**, **MK_FAST_NP**, **MK_ERRORCHECK**, **MK_RECURSIVE**, **MK_NORMAL**, or **MK_NOTSUP**.

Parameters

addrMutex attribute address.mutexattrCondition variable attribute handleprioceilingPointer to priority ceiling value.protocolpPointer to protocol value.psharedpPointer to pshared value.sessionSession handle.

Return Values

If successful, these functions return PTHDB SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB_BAD_MUTEXATTR Invalid mutex attribute handle. PTHDB BAD SESSION Invalid session handle. PTHDB_CALLBACK Debugger call back error. PTHDB_INTERNAL Error in library. PTHDB_NOSYS Not implemented PTHDB_POINTER Invalid pointer

Related Information

The **pthdebug.h** file.

The pthread.h file.

pthdb mutex addr, pthdb mutex lock count, pthdb mutex owner, pthdb mutex_pshared, pthdb mutex_prioceiling, pthdb mutex protocol, pthdb mutex state or pthdb mutex type **Subroutine**

Purpose

Gets the owner's pthread, mutex's pshared value, priority ceiling, protocol, lock state, and type.

Library

pthread debug library (libpthdebug.a)

Syntax

```
#include <sys/pthdebug.h>
int pthdb mutex addr (pthdb session t session,
                     pthdb_mutex_t mutex,
                     pthdb_addr_t * addrp)
int pthdb mutex owner (pthdb session t session,
                      pthdb mutex t
                                     mutex,
                      pthdb pthread t * ownerp)
int pthdb mutex lock count (pthdb session t session,
                           pthdb_mutex_t
                                            mutex,
                           int * countp);
int pthdb_mutex_pshared (pthdb_session_t session,
                        pthdb mutex t
                                         mutex,
                        pthdb_pshared_t * psharedp)
```

```
int pthdb mutex prioceiling (pthdb session t session,
                           pthdb mutex t mutex,
                           pthdb_pshared_t * prioceilingp)
int pthdb mutex protocol (pthdb session t session,
                         pthdb mutex t mutex,
                         pthdb pshared t * protocolp)
int pthdb_mutex_state (pthdb_session_t
                                          session,
                      pthdb mutex t
                                          mutex,
                      pthdb mutex state t * statep)
int pthdb mutex type (pthdb session t
                                        session,
                     pthdb mutex t
                                        mutex,
                     pthdb mutex type t * typep)
```

Description

pthdb_mutex_addr reports the address of the prhread_mutex_t.

pthdb_mutex_lock_count reports the lock count of the mutex.

pthdb_mutex_owner is used to get the pthread that owns the mutex.

The **pthdb_mutex_pshared** function is used to get the mutex process shared value. The pshared value can be **PSH_SHARED**, **PSH_PRIVATE**, or **PSH_NOTSUP**.

pthdb_mutex_prioceiling function is used to get the mutex priority ceiling value.

pthdb_mutex_protocol function is used to get the mutex protocol value. The protocol value can be **MP_INHERIT**, **MP_PROTECT**, **MP_NONE**, or **MP_NOTSUP**.

pthdb_mutex_state is used to get the value of the mutex lock state. The state can be MS_LOCKED,
MS_UNLOCKED or MS_NOTSUP.

pthdb_mutex_type is used to get the value of the mutex type. The values for the mutex type can be MK_NONRECURSIVE_NP, MK_RECURSIVE_NP, MK_FAST_NP, MK_ERRORCHECK, MK_RECURSIVE, MK_NORMAL, or MK_NOTSUP.

Parameters

addrMutex addresscountpMutex lock countmutexMutex handle

ownerpPointer to mutex ownerpsharedpPointer to pshared valueprioceilingpPointer to priority ceiling valueprotocolpPointer to protocol value

sessionSession handle.statepPointer to mutex statetypepPointer to mutex type

Return Values

If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB_BAD_MUTEX Invalid mutex handle. PTHDB BAD SESSION Invalid session handle. PTHDB_CALLBACK Debugger call back error. PTHDB_INTERNAL Call failed. PTHDB_NOSYS Not implemented PTHDB_POINTER Invalid pointer

Related Information

The pthdebug.h file and the pthread.h file.

The pthread.h file.

pthdb mutex waiter, pthdb cond waiter, pthdb rwlock read waiter or pthdb rwlock write waiter Subroutine

Purpose

Gets the next waiter in the list of an object's waiters.

Library

pthread debug library (libpthdebug.a)

Syntax

```
#include <sys/pthdebug.h>
int pthdb_mutex_waiter (pthdb_session_t session,
                        pthdb mutex t
                                        mutex,
                        pthdb pthread t * waiter,
                                         cmd);
int pthdb_cond_waiter (pthdb_session t session,
                        pthdb_cond_t
                                         cond,
                        pthdb_pthread_t * waiter,
                        int
                                         cmd)
int *pthdb rwlock read waiter (pthdb session t session,
                               pthdb rwlock t rwlock,
                               pthdb pthread t * waiter,
                                                cmd)
int *pthdb_rwlock_write_waiter (pthdb_session t session,
                                pthdb rwlock t rwlock,
                                pthdb pthread t * waiter,
                                int
                                                 cmd)
```

Description

The **pthdb mutex waiter** functions get the next waiter in the list of an object's waiters.

Each list is reset to the top of the list when the pthdb_session_update function is called, or when the list function reports a PTHDB INVALID * value. For example, when pthdb attr reports an attribute of PTHDB_INVALID_ATTR the list is reset to the beginning such that the next call reports the first attribute in the list, if any.

A report of PTHDB_INVALID_OBJECT represents the empty list or the end of a list, where OBJECT is one of these values: PTHREAD, ATTR, MUTEX, MUTEXATTR, COND, CONDATTR, RWLOCK, RWLOCKATTR, KEY, or TID as appropriate.

When PTHDB_LIST_FIRST is passed for the *cmd* parameter, the first item in the list is retrieved.

Parameters

sessionSession handle.mutexMutex object.

cond Condition variable object.

cmd Reset to the beginning of the list.

rwlock Read/Write lock object. waiter Pointer to waiter.

Return Values

If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB_BAD_SESSION Invalid session handle.
PTHDB_BAD_CMD Invalid command.
PTHDB_CALLBACK Debugger call back error.

PTHDB_INTERNAL Error in library.

PTHDB_MEMORY Not enough memory

PTHDB_POINTER Invalid pointer

Related Information

The **pthdebug.h** file.

The pthread.h file.

pthdb_pthread_arg Subroutine

Purpose

Reports the information associated with a pthread.

Library

pthread debug library (libpthdebug.a)

Syntax

```
int pthdb pthread cancelpend (pthdb session t session,
                             pthdb pthread t pthread,
                                              * cancelpendp)
int pthdb_pthread_cancelstate (pthdb_session t
                                                   session,
                              pthdb pthread t
                                                   pthread,
                              pthdb_cancelstate_t * cancelstatep)
int pthdb_pthread_canceltype (pthdb_session_t
                                                 session,
                             pthdb_pthread_t
                                                  pthread,
                             pthdb_canceltype_t * canceltypep)
int pthdb pthread detachstate (pthdb session t session,
                              pthdb_pthread_t pthread,
                              pthdb detachstate t * detachstatep)
int pthdb pthread exit (pthdb session t session,
                       pthdb pthread t pthread,
                       pthdb addr t
                                     * exitp)
int pthdb pthread func (pthdb session t session,
                        pthdb pthread t pthread,
                       pthdb addr t * funcp)
int pthdb_pthread_ptid (pthdb_session_t session,
                        pthdb pthread t pthread,
                       pthread t
                                      * ptidp)
int pthdb pthread schedparam (pthdb session t
                                                 session,
                             pthdb pthread t
                                                 pthread,
                             struct sched_param * schedparamp);
int pthdb_pthread_schedpolicy (pthdb_session_t session,
                              pthdb_pthread_t pthread,
                              pthdb_schedpolicy_t * schedpolicyp)
int pthdb_pthread_schedpriority (pthdb_session_t session,
                                 pthdb_pthread_t pthread,
                                                * schedpriorityp)
                                int
int pthdb_pthread_scope (pthdb_session_t session,
                        pthdb_pthread_t pthread,
                         pthdb_scope_t * scopep)
int pthdb pthread state (pthdb session t session,
                         pthdb pthread t pthread,
                         pthdb_state_t * statep)
int pthdb pthread_suspendstate (pthdb_session_t session,
                               pthdb pthread t pthread,
                               pthdb suspendstate t * suspendstatep)
int pthdb_ptid_pthread (pthdb_session_t session,
                       pthread t ptid,
                       pthdb pthread t
                                        * pthreadp)
```

Description

pthdb_pthread_arg reports the initial argument passed to the pthread's start function.

pthdb_pthread_addr reports the address of the pthread_t.

pthdb_pthread_cancelpend reports non-zero if cancellation is pending on the pthread; if not, it reports zero.

pthdb_pthread_cancelstate reports whether cancellation is enabled (**PCS_ENABLE**) or disabled (**PCS_DISABLE**). **PCS_NOTSUP** is reserved for unexpected results.

pthdb_pthread_canceltype reports whether cancellation is deferred (**PCT_DEFERRED**) or asynchronous (**PCT_ASYNCHRONOUS**). **PCT_NOTSUP** is reserved for unexpected results.

pthdb_pthread_detachstate reports whether the pthread is detached (**PDS_DETACHED**) or joinable (**PDS_JOINABLE**). **PDS_NOTSUP** is reserved for unexpected results.

pthdb_pthread_exit reports the exit status returned by the pthread via **pthread_exit**. This is only valid if the pthread has exited (**PST_TERM**).

pthdb pthread func reports the address of the pthread's start function.

pthdb_pthread_ptid reports the pthread identifier (pthread_t) associated with the pthread.

pthdb_pthread_schedparam reports the pthread's scheduling parameters. This currently includes policy and priority.

pthdb_pthread_schedpolicy reports whether the pthread's scheduling policy is other (**SP_OTHER**), first in first out (**SP_FIFO**), or round robin (**SP_RR**). **SP_NOTSUP** is reserved for unexpected results.

pthdb_pthread_schedpriority reports the pthread's scheduling priority.

pthdb_pthread_scope reports whether the pthread has process scope (**PS_PROCESS**) or system scope (**PS_SYSTEM**). **PS_NOTSUP** is reserved for unexpected results.

pthdb_pthread_state reports whether the pthread is being created (**PST_IDLE**), currently running (**PST_RUN**), waiting on an event (**PST_SLEEP**), waiting on a cpu (**PST_READY**), or waiting on a join or detach (**PST_TERM**). **PST_NOTSUP** is reserved for unexpected results.

pthdb_pthread_suspendstate reports whether the pthread is suspended (**PSS_SUSPENDED**) or not (**PSS_UNSUSPENDED**). **PSS_NOTSUP** is reserved for unexpected results.

pthdb_ptid_pthread reports the pthread for the ptid.

Parameters

addr pthread address Initial argument buffer. argp Cancel pending buffer. cancelpendp cancelstatep Cancel state buffer. canceltypep Cancel type buffer. Detach state buffer. detachstatep exitp Exit value buffer. Start function buffer. funcp pthread pthread handle.

pthreadp Pointer to pthread handle.

ptid pthread identifier ptidp pthread identifier buffer. schedparamp Scheduling parameters buffer. schedpolicyp Scheduling policy buffer. Scheduling priority buffer. schedpriorityp Contention scope buffer. scopep

session Session handle. statep State buffer.

suspendstatep Suspend state buffer.

Return Values

If successful, these functions return PTHDB_SUCCESS, else an error code is returned.

Error Codes

PTHDB_BAD_SESSION Invalid session handle. PTHDB_BAD_PTHREAD Invalid pthread handle. PTHDB_BAD_POINTER Invalid buffer pointer.

PTHDB_BAD_PTID Invalid ptid.

PTHDB_CALLBACK Debugger call back error.

PTHDB_NOTSUP Not supported. PTHDB INTERNAL Error in library.

Related Information

The pthdebug.h file.

The pthread.h file.

pthdb_pthread_context or pthdb_pthread_setcontext Subroutine

Purpose

Provides access to the pthread context via the struct context64 structure.

Library

pthread debug library (libpthdebug.a)

Syntax

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

```
int pthdb pthread context (pthdb session t
                                             session,
                          pthdb_pthread_t pthread,
                           pthdb context t * context)
int pthdb_pthread_setcontext (pthdb_session_t
                                               session,
                             pthdb pthread t
                                               pthread,
                             pthdb context t * context)
```

Description

The pthread debug library provides access to the pthread context via the struct context64 structure, whether the process is 32-bit or 64-bit. The debugger should be able to convert from 32-bit to 64-bit and from 64-bit for 32-bit processes. The extent to which this structure is filled in depends on the presence of the PTHDB FLAG GPRS, PTHDB FLAG SPRSI and PTHDB FLAG FPRS session flags. It is necessary to use the pthread debug library to access the context of a pthread without a kernel thread. The pthread debug library can also be used to access the context of a pthread with a kernel thread, but this results in a call back to the debugger, meaning that the debugger is capable of obtaining this information by itself. The debugger determines if the kernel thread is running in user mode or kernel mode and then fills in the struct context64 appropriately. The pthread debug library does not use this information itself and is thus not sensitive to the correct implementation of the read regs and write regs call back functions.

pthdb_pthread_context reports the context of the pthread based on the settings of the session flags. Uses the read regs call back if the pthread has a kernel thread. If read regs is not defined, then it returns **PTHDB NOTSUP**.

pthdb_pthread_setcontext sets the context of the pthread based on the settings of the session flags. Uses the write_data call back if the pthread does not have a kernel thread. Use the write_regs call back if the pthread has a kernel thread.

If the debugger does not define the read_regs and write_regs call backs and if the pthread does not have a kernel thread, then the pthdb pthread context and pthdb pthread setcontext functions succeed. But if a pthread does not have a kernel thread, then these functions fail and return PTHDB CONTEXT.

Parameters

Session handle. session pthread handle. pthread context Context buffer pointer.

Return Values

If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB BAD SESSION Invalid session handle. PTHDB BAD PTHREAD Invalid pthread handle. PTHDB_BAD_POINTER Invalid buffer pointer. PTHDB CALLBACK Callback function failed.

PTHDB CONTEXT Could not determine pthread context.

PTHDB MEMORY Not enough memory

pthdb_pthread_(set)context returns PTHDB_NOTSUP if PTHDB_NOTSUP

the read_regs, write_data or write_regs call backs are

set to NULL.

PTHDB_INTERNAL Error in library.

Related Information

The pthdebug.h file.

The pthread.h file.

pthdb_pthread_hold, pthdb_pthread_holdstate or pthdb_pthread_unhold Subroutine

Purpose

Reports and changes the hold state of the specified pthread.

Library

pthread debug library (libpthdebug.a)

Syntax

Description

pthdb_pthread_holdstate reports if a pthread is held. The possible hold states are PHS_HELD,
PHS_NOTHELD, or PHS_NOTSUP.

pthdb_pthread_hold prevents the specified pthread from running.

pthdb_pthread_unhold unholds the specified pthread. The pthread held earlier can be unheld by calling this function.

Notes:

- 1. You must always use the **pthdb_pthread_hold** and **pthdb_pthread_unhold** functions, regardless of whether or not a pthread has a kernel thread.
- 2. These functions are only supposted when the PTHDB_FLAG_HOLD is set.

Parameters

session Session handle.

pthread handle. The specified pthread should have an attached kernel thread

id.

holdstatep Pointer to the hold state

Return Values

If successful, pthdb_pthread_hold returns PTHDB_SUCCESS. Otherwise, it returns an error code.

Error Codes

PTHDB_BAD_PTHREAD Invalid pthread handle.
PTHDB_BAD_SESSION Invalid session handle.
PTHDB_HELD pthread is held.
PTHDB_INTERNAL Error in library.

Related Information

The pthdb_session_setflags subroutine.

The **pthdebug.h** file.

The pthread.h file.

pthdb_pthread_sigmask, pthdb_pthread_sigpend or pthdb_pthread_sigwait Subroutine

Purpose

Returns the pthread signals pending, the signals blocked, the signals received, and awaited signals.

Library

pthread debug library (libpthdebug.a)

Syntax

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

```
int pthdb_pthread_sigmask (pthdb_session_t session,
                          pthdb_pthread_t pthread,
                          sigset t
                                        * sigsetp)
int pthdb_pthread_sigpend (pthdb_session_t session,
                          pthdb pthread t pthread,
                          sigset t
                                       * sigsetp)
int pthdb pthread sigwait (pthdb session t session,
                          pthdb_pthread_t pthread,
                          sigset t
                                         * sigsetp)
```

Description

pthdb_pthread_sigmask reports the signals that the pthread has blocked.

pthdb_pthread_sigpend reports the signals that the pthread has pending.

pthdb pthread sigwait reports the signals that the pthread is waiting on.

Parameters

session	Session handle.
pthread	Pthread handle
sigsetp	Signal set buffer.

Return Values

If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Code

```
PTHDB_BAD_SESSION
                                         Invalid session handle.
PTHDB_BAD_PTHREAD
                                         Invalid pthread handle.
PTHDB BAD POINTER
                                         Invalid buffer pointer.
PTHDB CALLBACK
                                         Debugger call back error.
```

Related Information

The **pthdebug.h** file.

The pthread.h file.

pthdb_pthread_specific Subroutine

Purpose

Reports the value associated with a pthreads specific data key.

Library

pthread debug library (libpthdebug.a)

Syntax

```
#include <sys/pthdebug.h>
void *pthdb pthread specific(pthdb session t session,
                           pthdb pthread t pthread,
                           pthdb key t key,
                           pthdb addr t * specificp)
```

Description

Each process has active pthread specific data keys. Each active pthread specific data key is in use by one or more pthreads. Each pthread can have its own value associated with each pthread specific data key. The **pthdb_pthread_specific** function provide access to those values.

pthdb_pthread_specific reports the specific data value for the pthread and key combination.

Parameters

session The session handle. pthread The pthread handle.

key The key.

specificp Specific data value buffer.a

Return Values

If successful, pthdb_pthread_specific returns PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB BAD SESSION Invalid session handle. PTHDB_BAD_PTHREAD Invalid pthread handle.

PTHDB_BAD_KEY Invalid key.

PTHDB_BAD_POINTER Invalid buffer pointer. PTHDB_CALLBACK Debugger call back error.

PTHDB_INTERNAL Error in library.

Related information

The pthdebug.h file.

The pthread.h file.

pthdb_pthread_tid or pthdb_tid_pthread Subroutine

Purpose

Gets the kernel thread associated with the pthread and the pthread associated with the kernel thread.

Library

pthread debug library (libpthdebug.a)

Syntax

Description

pthdb_pthread_tid gets the kernel thread id associated with the pthread.

pthdb_tid_pthread is used to get the pthread associated with the kernel thread.

Parameters

sessionSession handle.pthreadPthread handle

pthreadp Pointer to pthread handle

tid Kernel thread id

tidp Pointer to kernel thread id

Return Values

If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB_BAD_PTHREAD Invalid pthread handle.
PTHDB_BAD_SESSION Invalid session handle.

PTHDB_BAD_TID Invalid tid.

PTHDB_CALLBACK Debugger call back error.

PTHDB_INTERNAL Error in library.

PTHDB_INVALID_TID Empty list or the end of a list.

Related Information

The pthdebug.h file.

The pthread.h file.

pthdb_rwlockattr_addr, or pthdb_rwlockattr_pshared Subroutine

Purpose

Gets the rwlock attribute pshared values.

Library

pthread debug library (libpthdebug.a)

Syntax

Description

pthdb_rwlockattr_addr reports the address of the pthread_rwlockattr_t.

pthdb_rwlockattr_pshared is used to get the rwlock attribute process shared value. The pshared value can be **PSH_SHARED**, **PSH_PRIVATE**, or **PSH_NOTSUP**.

Parameters

addrRead/Write lock attribute address.psharedpPointer to the pshared value.rwlockattrRead/Write lock attribute handlesessionSession handle.

Return Values

If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB_BAD_RWLOCKATTR
Invalid rwlock attribute handle.
PTHDB_BAD_SESSION
Invalid session handle.
PTHDB_CALLBACK
PTHDB_INTERNAL
PTHDB_POINTER
Invalid pointer

Related Information

The **pthdebug.h** file.

The pthread.h file.

pthdb_rwlock_addr, pthdb_rwlock_lock_count, pthdb_rwlock_owner, pthdb rwlock pshared or pthdb rwlock state Subroutine

Purpose

Gets the owner, the pshared value, or the state of the read/write lock.

Library

pthread debug library (libpthdebug.a)

Syntax

```
#include <sys/pthdebug.h>
int pthdb rwlock addr (pthdb session t session,
                      pthdb_rwlock_t rwlock,
                      pthdb_addr_t * addrp)
int pthdb_rwlock_lock_count (pthdb_session_t session,
                            pthdb rwlock t
                                             rwlock,
                            int * countp);
int pthdb rwlock owner (pthdb session t session,
                       pthdb_rwlock_t rwlock,
                       pthdb pthread t * ownerp
                       int
                                        cmd)
int pthdb rwlock pshared (pthdb session t session,
                         pthdb rwlock t rwlock,
                         pthdb pshared t * psharedp)
int pthdb rwlock state (pthdb session t
                                          session,
                       pthdb rwlock t
                                           rwlock,
                       pthdb_rwlock_state_t * statep)
```

Description

The **pthdb_rwlock_addr** function reports the address of the pthdb_rwlock_t.

The **pthdb_rwlock_lock_count** function reports the lock count for the rwlock.

The pthdb_rwlock_owner function is used to get the read/write lock owner's pthread handle.

The pthdb_rwlock_pshared function is used to get the rwlock attribute process shared value. The pshared value can be PSH_SHARED, PSH_PRIVATE, or PSH_NOTSUP.

The pthdb_rwlock_state is used to get the read/write locks state. The state can be RWLS_NOTSUP, RWLS_WRITE, RWLS_FREE, and RWLS_READ.

Parameters

addrp Read write lock address.

countp Read write lock lock count.

cmd can be PTHDB_LIST_FIRST to get the first owner in

the list of owners or **PTHDB_LIST_NEXT** to get the next owner in the list of owners. The list is empty or ended by

*owner == PTHDB_INVALID_PTHREAD.

ownerp Pointer to pthread which owns the rwlock

psharedpPointer to pshared valuerwlockRead write lock handlesessionSession handle.

sessionSession handle.statepPointer to state value

Return Values

If successful, these functions return PTHDB_SUCCESS. Otherwise, an error code is returned.

Error Codes

PTHDB_BAD_SESSION Invalid session handle.
PTHDB_BAD_CMD Invalid command passed.
PTHDB_CALLBACK Debugger call back error.

PTHDB_INTERNALError in library.PTHDB_POINTERInvalid pointer

Related Information

The pthdebug.h file.

The **pthread.h** file.

pthdb_session_committed Subroutines

Purpose

Facilitates examining and modifying multi-threaded application's pthread library object data.

Library

pthread debug library (libpthdebug.a)

Syntax

```
pthdb callbacks t * callbacks,
                       pthdb session t * sessionp)
int pthdb_session_pthreaded (pthdb_user_t user,
                            unsigned long long flags
                            pthdb callbacks t * callbacks,
                                             ** name)
int pthdb session continue tid (pthdb session t session,
                               tid t
                                              * tidp,
                               int
                                               cmd);
int pthdb session stop tid (pthdb session t session,
                           tid t
int pthdb_session_commit_tid (pthdb_session_t session,
                             tid t
                                   * tidp.
                                              cmd);
                             int
int pthdb session setflags (pthdb session t
                                              session,
                           unsigned long long flags)
int pthdb session update (pthdb session t session)
```

Description

To facilitate debugging multiple processes, the pthread debug library supports multiple sessions, one per process. Functions are provided to initialize, destroy, and customize the behavior of these sessions. In addition, functions are provided to query global fields of the pthread library. All functions in the library require a session handle associated with an initialized session except pthdb session init, which initializes sessions, and pthdb session pthreaded, which can be called before the session has been initialized.

pthdb session committed reports the symbol name of a function called after the hold/unhold commit operation has completed. This symbol name can be used to set a breakpoint to notify the debugger when the hold/unhold commit has completed. The actual symbol name reported may change at any time. The function name returned is implemented in assembly with the following code:

```
ori 0.0. 0
                   # no-op
                   # return to caller
blr.
```

This allows the debugger to overwrite the no-op with a trap instruction and leave it there by stepping over it. This function is only supported when the PTHDB_FLAG_HOLD flag is set.

pthdb_session_concurrency reports the concurrency level of the pthread library. The concurrency level is the M:N ratio, where N is always 1.

pthdb_session_destroy notifies the pthread debug library that the debugger or application is finished with the session. This deallocates any memory associated with the session and allows the session handle to be reused.

pthdb_session_setflags changes the flags for a session. With these flags, a debugger can customize the session. Flags consist of the following values or-ed together:

PTHDB_FLAG_GPRS	The general purpose registers should be included in any context read or write, whether internal to the library or via call backs to the debugger.
PTHDB_FLAG_SPRS	The special purpose registers should be included in any context read or write whether internal to the library or via call backs to the debugger.
PTHDB_FLAG_FPRS	The floating point registers should be included in any context read or write whether internal to the library or via call backs to the debugger.
PTHDB_FLAG_REGS	All registers should be included in any context read or write whether internal to the library or via call backs to the debugger. This is equivalent to PTHDB_FLAG_GPRSIPTHDB_FLAG_GPRSIPTHDB_FLAG_GPRSIPTHDB_FLAG_GPRSIPTHDB

PTHDB_FLAG_HOLD The debugger will be using the pthread debug library hold/unhold facilities to

prevent the execution of pthreads. This flag cannot be used with

PTHDB_FLAG_SUSPEND. This flag should be used by debuggers, only.

Applications will be using the pthread library suspend/continue facilities to

prevent the execution of pthreads. This flag cannot be used with

PTHDB_FLAG_HOLD. This flag is for introspective mode and should be used by applications, only

by applications, only.

Note: PTHDB_FLAG_HOLD and PTHDB_FLAG_SUSPEND can only be passed to the **pthdb_session_init** function. Neither PTHDB_FLAG_HOLD nor PTHDB_FLAG_SUSPEND should be passed to **pthdb_session_init** when

debugging a core file.

The **pthdb_session_flags** function gets the current flags for the session.

The **pthdb_session_init** function tells the pthread debug library to initialize a session associated with the unique given user handle. **pthdb_session_init** will assign a unique session handle and return it to the debugger. If the application's execution mode is 32 bit, then the debugger should initialize the **exec_mode** to **PEM_32BIT**. If the application's execution mode is 64 bit, then the debugger should initialize **mode** to **PEM_64BIT**. The **flags** are documented above with the **pthdb_session_setflags** function. The **callback** parameter is a list of call back functions. (Also see the **pthdebug.h** header file.) The **pthdb_session_init** function calls the **symbol_addrs** function to get the starting addresses of the symbols and initializes these symbols' starting addresses within the pthread debug library.

pthdb_session_pthreaded reports the symbol name of a function called after the pthread library has been initialized. This symbol name can be used to set a breakpoint to notify the debugger when to initialize a pthread debug library session and begin using the pthread debug library to examine pthread library state. The actual symbol name reported may change at any time. This function, is the only pthread debug library function that can be called before the pthread library is initialized. The function name returned is implemented in assembly with the following code:

```
ori 0,0,0 # no-op
blr # return to caller
```

This is conveniently allows the debugger to overwrite the no-op with a trap instruction and leave it there by stepping over it.

The **pthdb_session_continue_tid** function allows the debugger to obtain the list of threads that must be continued before it proceeds with single stepping a single pthread or continuing a group of pthreads. This function reports one tid at a time. If the list is empty or the end of the list has been reached, **PTHDB_INVALID_TID** is reported. The debugger will need to continue any pthreads with kernel threads that it wants. The debugger is responsible for parking the stop thread and continuing the stop thread. The *cmd* parameter can be either **PTHDB_LIST_NEXT** or **PTHDB_LIST_FIRST**; if **PTHDB_LIST_FIRST** is passed, then the internal counter will be reset and the first tid in the list will be reported.

Note: This function is only supported when the PTHDB_FLAG_HOLD flag is set.

The **pthdb_session_stop_tid** function informs the pthread debug library, which informs the pthread library the tid of the thread that stopped the debugger.

Note: This function is only supported when the **PTHDB_FLAG_HOLD** flag is set.

pthdb_session_commit_tid reports subsequent kernel thread identifiers which must be continued to commit the hold and unhold changes. This function reports one tid at a time. If the list is empty or the end of the list has been reached, PTHDB_INVALID_TID is reported. The cmd parameter can be either PTHDB_LIST_NEXT or PTHDB_LIST_FIRST, if PTHDB_LIST_FIRST is passed then the internal counter will be reset and first tid in the list will be reported.

Note: This function is only supported when the PTHDB FLAG HOLD flag is set.

pthdb_session_update tells the pthread debug library to update it's internal information concerning the state of the pthread library. This should be called each time the process stops before any other pthread debug library functions to ensure their results are reliable.

Each list is reset to the top of the list when the pthdb_session_update function is called, or when the list function reports a PTHDB_INVALID_* value. For example, when pthdb_attr reports an attribute of PTHDB_INVALID_ATTR the list is reset to the beginning such that the next call reports the first attribute in the list, if any.

A report of **PTHDB INVALID** OBJECT represents the empty list or the end of a list, where OBJECT is one of these values: PTHREAD, ATTR, MUTEX, MUTEXATTR, COND, CONDATTR, RWLOCK, RWLOCKATTR, KEY, or TID as appropriate.

Parameters

session Session handle. user Debugger user handle. Pointer to session handle. sessionp name Symbol name buffer.

cmd Reset to the beginning of the list.

concurrencyp Library concurrency buffer.

flags Session flags.

Pointer to session flags. flagsp Debuggee execution mode: exec_mode

PEM_32BIT for 32-bit processes or PEM_64BIT for 64-bit processes.

callbacks Call backs structure. tid Kernel thread id. tidp Kernel thread id buffer...

Return Values

If successful, these functions return PTHDB SUCCESS. Otherwise, they return an error value.

Error Codes

Invalid session handle. PTHDB_BAD_SESSION

PTHDB BAD VERSION Invalid pthread debug library or pthread library version.

PTHDB_BAD_MODE Invalid execution mode. PTHDB BAD FLAGS Invalid session flags.

PTHDB BAD CALLBACK Insufficient call back functions.

Invalid command. PTHDB_BAD_CMD PTHDB BAD POINTER Invalid buffer pointer. PTHDB BAD USER Invalid user handle. PTHDB CALLBACK Debugger call back error. PTHDB_MEMORY Not enough memory. PTHDB NOSYS Function not implemented.

PTHDB_NOT_PTHREADED pthread library not initialized. pthread library symbol not found. PTHDB SYMBOL

PTHDB_INTERNAL Error in library.

Related Information

The pthdebug.h file.

The pthread.h file.

pthread atfork Subroutine

Purpose

Registers fork handlers.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <sys/types.h>
#include <unistd.h>
int pthread atfork (prepare, parent, child)
void (*prepare)(void);
void (*parent)(void);
void (*child)(void);
```

Description

The pthread_atfork subroutine registers fork cleanup handlers. The prepare handler is called before the processing of the **fork** subroutine commences. The parent handler is called after the processing of the fork subroutine completes in the parent process. The child handler is called after the processing of the fork subroutine completes in the child process.

When the **fork** subroutine is called, only the calling thread is duplicated in the child process, but all synchronization variables are duplicated. The pthread_atfork subroutine provides a way to prevent state inconsistencies and resulting deadlocks. The expected usage is that the prepare handler acquires all mutexes, and the two other handlers release them in the parent and child processes.

The prepare handlers are called in LIFO (Last In First Out) order; whereas the parent and child handlers are called in FIFO (first-in first-out) order. Thereafter, the order of calls to the pthread atfork subroutine is significant.

Note: The pthread.h header file must be the first included file of each source file using the threads library.

Parameters

Points to the pre-fork cleanup handler. If no pre-fork handling is desired, the value of this pointer should prepare

be set to NULL.

Points to the parent post-fork cleanup handler. If no parent post-fork handling is desired, the value of parent

this pointer should be set to NULL.

child Points to the child post-fork cleanup handler. If no child post-fork handling is desired, the value of this

pointer should be set to **NULL**.

Return Values

Upon successful completion, the **pthread atfork** subroutine returns a value of zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_atfork subroutine will fail if:

ENOMEM Insufficient table space exists to record the fork handler addresses.

The pthread atfork subroutine will not return an error code of EINTR.

Related Information

The **fork** ("fork, f_fork, or vfork Subroutine" on page 248) subroutine, **atexit** ("exit, atexit, _exit, or _Exit Subroutine" on page 203) subroutine.

Process Duplication and Termination in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_destroy Subroutine

Purpose

Deletes a thread attributes object.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread_attr_destroy (attr)
pthread_attr_t *attr;
```

Description

The **pthread_attr_destroy** subroutine destroys the thread attributes object *attr*, reclaiming its storage space. It has no effect on the threads previously created with that object.

Parameters

attr Specifies the thread attributes object to delete.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_attr_destroy subroutine is unsuccessful if the following is true:

EINVAL The *attr* parameter is not valid.

This function will not return an error code of [EINTR].

Related Information

The **pthread_attr_init** ("pthread_attr_init Subroutine" on page 911) subroutine, **pthread_create** ("pthread_create Subroutine" on page 930) subroutine, the **pthread.h** file.

Creating Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_getguardsize or pthread_attr_setguardsize Subroutines

Purpose

Gets or sets the thread guardsize attribute.

Library

Threads Library (libthreads.a)

Syntax

```
#include <pthread.h>
int pthread_attr_getguardsize (attr, guardsize)
const pthread_attr_t *attr;
size t *guardsize;
int pthread attr setguardsize (attr, guardsize)
pthread attr t *attr;
size t guardsize;
```

Description

The *guardsize* attribute controls the size of the guard area for the created thread's stack. The *guardsize* attribute provides protection against overflow of the stack pointer. If a thread's stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error results (possibly in a SIGSEGV signal being delivered to the thread).

The guardsize attribute is provided to the application for two reasons:

- · Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads will never overflow their stack, can save system resources by turning off guard areas.
- When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

The pthread attr getguardsize function gets the guardsize attribute in the attr object. This attribute is returned in the *guardsize* parameter.

The pthread_attr_setguardsize function sets the guardsize attribute in the attr object. The new value of this attribute is obtained from the *quardsize* parameter. If *quardsize* is zero, a quard area will not be provided for threads created with attr. If guardsize is greater than zero, a guard area of at least size guardsize bytes is provided for each thread created with attr.

A conforming implementation is permitted to round up the value contained in *quardsize* to a multiple of the configurable system variable PAGESIZE (see sys/mman.h). If an implementation rounds up the value of guardsize to a multiple of PAGESIZE, a call to pthread_attr_getguardsize specifying attr will store in the guardsize parameter the guard size specified by the previous pthread_attr_setguardsize function call. The default value of the guardsize attribute is PAGESIZE bytes. The actual value of PAGESIZE is implementation-dependent and may not be the same on all implementations.

If the stackaddr attribute has been set (that is, the caller is allocating and managing its own thread stacks), the guardsize attribute is ignored and no protection will be provided by the implementation. It is the responsibility of the application to manage stack overflow along with stack allocation and management in this case.

Parameters

attr Specifies the thread attributes object.

guardsize Controls the size of the guard area for the created thread's stack, and protects against

overflow of the stack pointer.

Return Values

If successful, the **pthread_attr_getguardsize** and **pthread_attr_setguardsize** functions return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_attr_getguardsize and pthread_attr_setguardsize functions will fail if:

EINVAL The attribute *attr* is invalid.

EINVAL The *guardsize* parameter is invalid.

EINVAL The *guardsize* parameter contains an invalid value.

pthread_attr_getschedparam Subroutine

Purpose

Returns the value of the schedparam attribute of a thread attributes object.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
#include <sys/sched.h>
int pthread_attr_getschedparam (attr, schedparam)
const pthread_attr_t *attr;
struct sched param *schedparam;
```

Description

The **pthread_attr_getschedparam** subroutine returns the value of the schedparam attribute of the thread attributes object *attr*. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The sched_priority field of the **sched_param** structure contains the priority of the thread. It is an integer value.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr Specifies the thread attributes object.

Points to where the schedparam attribute value will be stored.

Return Values

Upon successful completion, the value of the schedparam attribute is returned via the schedparam parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread attr getschedparam subroutine is unsuccessful if the following is true:

EINVAL The attr parameter is not valid.

This function does not return EINTR.

Related Information

The pthread_attr_setschedparam ("pthread_attr_setschedparam Subroutine" on page 915) subroutine, pthread attr init ("pthread attr init Subroutine" on page 911) subroutine, pthread getschedparam ("pthread getschedparam Subroutine" on page 940) subroutine, the pthread.h file.

Threads Scheduling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_getstackaddr Subroutine

Purpose

Returns the value of the stackaddr attribute of a thread attributes object.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

```
int pthread attr getstackaddr (attr, stackaddr)
const pthread_attr_t *attr;
void **stackaddr;
```

Description

The pthread_attr_getstackaddr subroutine returns the value of the stackaddr attribute of the thread attributes object attr. This attribute specifies the stack address of the thread created with this attributes object.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr Specifies the thread attributes object.

Points to where the stackaddr attribute value will be stored. stackaddr

Return Values

Upon successful completion, the value of the stackaddr attribute is returned via the stackaddr parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_attr_getstackaddr subroutine is unsuccessful if the following is true:

EINVAL The attr parameter is not valid.

This function will not return EINTR.

Related Information

The pthread_attr_setstackaddr ("pthread_attr_setstackaddr Subroutine" on page 916) subroutine, pthread attr init ("pthread attr init Subroutine" on page 911) subroutine, the pthread.h file.

Advanced Attributes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_getstacksize Subroutine

Purpose

Returns the value of the stacksize attribute of a thread attributes object.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

int pthread attr getstacksize (attr, stacksize) const pthread attr t *attr; size t *stacksize;

Description

The pthread_attr_getstacksize subroutine returns the value of the stacksize attribute of the thread attributes object attr. This attribute specifies the minimum stacksize of a thread created with this attributes object. The value is given in bytes. For 32-bit compiled applications, the default stacksize is 96 KB (defined in the pthread.h file). For 64-bit compiled applications, the default stacksize is 192 KB (defined in the pthread.h file).

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

Parameters

attr Specifies the thread attributes object.

Points to where the stacksize attribute value will be stored. stacksize

Return Values

Upon successful completion, the value of the stacksize attribute is returned via the stacksize parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread attr getstacksize subroutine is unsuccessful if the following is true:

EINVAL The attr or stacksize parameters are not valid.

This function will not return an error code of [EINTR].

Related Information

The pthread_attr_setstacksize ("pthread_attr_setstacksize Subroutine" on page 917) subroutine, pthread attr init ("pthread attr init Subroutine") subroutine, the pthread.h file.

Advanced Attributes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_init Subroutine

Purpose

Creates a thread attributes object and initializes it with default values.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
```

```
int pthread_attr_init ( attr)
pthread_attr_t *attr;
```

Description

The pthread attr init subroutine creates a new thread attributes object attr. The new thread attributes object is initialized with the following default values:

Always initialized

Attribute	Default value
Detachstate	PTHREAD_CREATE_JOINABLE
	PTHREAD_SCOPE_PROCESS the default ensures compatibility with implementations that do not support this POSIX option.

Always initialized

Attribute	Default value
Inheritsched	PTHREAD_INHERITSCHED
Schedparam	A sched_param structure which sched_prio field is set to 1, the least favored priority.
Schedpolicy	SCHED_OTHER
Stacksize	PTHREAD_STACK_MIN
Guardsize	PAGESIZE

The resulting attribute object (possibly modified by setting individual attribute values), when used by pthread_create, defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to pthread_create.

Parameters

attr Specifies the thread attributes object to be created.

Return Values

Upon successful completion, the new thread attributes object is filled with default values and returned via the attr parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_init** subroutine is unsuccessful if the following is true:

EINVAL The attr parameter is not valid.

ENOMEM There is not sufficient memory to create the thread attribute object.

This function will not return an error code of [EINTR].

Related Information

The pthread attr setdetachstate ("pthread attr getdetachstate or pthread attr setdetachstate Subroutines" on page 913) subroutine, pthread_attr_setstackaddr ("pthread_attr_setstackaddr Subroutine" on page 916) subroutine, pthread attr setstacksize ("pthread attr setstacksize Subroutine" on page 917) subroutine, pthread create ("pthread create Subroutine" on page 930) subroutine, pthread attr destroy ("pthread attr destroy Subroutine" on page 906) subroutine. pthread_attr_setguardsize ("pthread_attr_getguardsize or pthread_attr_setguardsize Subroutines" on page 907) subroutine.

The pthread.h file.

Creating Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_getdetachstate or pthread_attr_setdetachstate **Subroutines**

Purpose

Sets and returns the value of the detachstate attribute of a thread attributes object.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread attr setdetachstate (attr, detachstate)
pthread attr t *attr;
int detachstate;
int pthread_attr_getdetachstate (attr, detachstate)
const pthread_attr_t *attr;
int *detachstate;
```

Description

The detachstate attribute controls whether the thread is created in a detached state. If the thread is created detached, then use of the ID of the newly created thread by the pthread detach or pthread join function is an error.

The pthread_attr_setdetachstate and pthread_attr_getdetachstate, respectively, set and get the detachstate attribute in the attr object.

The detachstate attribute can be set to either PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE. A value of PTHREAD_CREATE_DETACHED causes all threads created with attr to be in the detached state, whereas using a value of PTHREAD_CREATE_JOINABLE causes all threads created with attr to be in the joinable state. The default value of the detachstate attribute is PTHREAD_CREATE_JOINABLE.

Parameters

Specifies the thread attributes object.

Points to where the detachstate attribute value will be stored. detachstate

Return Values

Upon successful completion, pthread attr setdetachstate and pthread attr getdetachstate return a value of **0**. Otherwise, an error number is returned to indicate the error.

The pthread_attr_getdetachstate function stores the value of the detachstate attribute in the detachstate parameter if successful.

Error Codes

The pthread_attr_setdetachstate function will fail if:

EINVAL The value of detachstate was not valid. The pthread_attr_getdetachstate and pthread_attr_setdetachstate functions will fail if:

EINVAL The attribute parameter is invalid.

These functions will not return an error code of EINTR.

Related Information

The "pthread attr setstackaddr Subroutine" on page 916, "pthread_attr_setstacksize Subroutine" on page 917, "pthread create Subroutine" on page 930, and "pthread attr init Subroutine" on page 911.

The pthread.h file in AIX 5L Version 5.2 Files Reference

Creating Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread attr getscope and pthread attr setscope Subroutines

Purpose

Gets and sets the scope attribute in the attr object.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread_attr_setscope (attr, contentionscope)
pthread_attr_t *attr;
int contentionscope;
int pthread_attr_getscope (attr, contentionscope)
const pthread attr t *attr;
int *contentionscope;
```

Description

The scope attribute controls whether a thread is created in system or process scope.

The pthread attr getscope and pthread attr setscope subroutines get and set the scope attribute in the attr object.

The scope can be set to PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS. A value of PTHREAD_SCOPE_SYSTEM causes all threads created with the attr parameter to be in system scope, whereas a value of PTHREAD SCOPE PROCESS causes all threads created with the attr parameter to be in process scope.

The default value of the contentionscope parameter is PTHREAD_SCOPE_PROCESS.

Parameters

attr Specifies the thread attributes object.

contentionscope Points to where the scope attribute value will be stored.

Return Values

Upon successful completion, the pthread_attr_getscope and pthread_attr_setscope subroutines return a value of 0. Otherwise, an error number is returned to indicate the error.

Error Codes

EINVAL The value of the attribute being set/read is not valid.

ENOTSUP An attempt was made to set the attribute to an unsupported value.

Related Information

The "pthread create Subroutine" on page 930, and "pthread attr init Subroutine" on page 911.

The pthread.h file in AIX 5L Version 5.2 Files Reference

Creating Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_setschedparam Subroutine

Purpose

Sets the value of the schedparam attribute of a thread attributes object.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
#include <sys/sched.h>
int pthread attr setschedparam (attr, schedparam)
pthread attr t *attr;
const struct sched param *schedparam;
```

Description

The pthread_attr_setschedparam subroutine sets the value of the schedparam attribute of the thread attributes object attr. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The sched priority field of the sched_param structure contains the priority of the thread.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr Specifies the thread attributes object.

Points to where the scheduling parameters to set are stored. The sched priority field must be in schedparam

the range from 1 to 127, where 1 is the least favored priority, and 127 the most favored.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_attr_setschedparam subroutine is unsuccessful if the following is true:

EINVAL The attr parameter is not valid.

ENOSYS The priority scheduling POSIX option is not implemented. **ENOTSUP** The value of the schedparam attribute is not supported.

Related Information

The pthread attr getschedparam ("pthread attr getschedparam Subroutine" on page 908) subroutine, pthread_attr_init ("pthread_attr_init Subroutine" on page 911) subroutine, pthread_create ("pthread_create Subroutine" on page 930) subroutine, the pthread.h file.

Threads Scheduling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread attr setstackaddr Subroutine

Purpose

Sets the value of the stackaddr attribute of a thread attributes object.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h> int pthread_attr_setstackaddr (attr, stackaddr) pthread attr t *attr; void *stackaddr;

Description

The pthread_attr_setstackaddr subroutine sets the value of the stackaddr attribute of the thread attributes object attr. This attribute specifies the stack address of a thread created with this attributes object.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

A Provision has been made in libpthreads to create guardpages for the user stack internally. This is used for debugging purposes only. By default, it is turned off and can be invoked by exporting the following environment variable:

AIXTHREAD GUARDPAGES FOR USER STACK=n (Where n is the decimal number of guard pages.)

Note: Even if it is exported, guard pages will only be constructed if both the stackaddr and stacksize attributes have been set by the caller for the thread. Also, the guard pages and alignment pages will be created out of the user's stack (which will reduce the stack size). If the new stack size after creating guard pages is less than the minimum stack size (PTHREAD STACK MIN), then the guard pages will not be constructed.

Parameters

attr Specifies the thread attributes object.

stackaddr Specifies the stack address to set. It is a void pointer. The address that needs to be passed is not

the beginning of the malloc generated address but the beginning of the stack. For example:

stackaddr = malloc(stacksize); pthread_attr_setstackaddr(&thread, stackaddr + stacksize);

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread attr setstackaddr subroutine is unsuccessful if the following is true:

EINVAL The attr parameter is not valid.

ENOSYS The stack address POSIX option is not implemented.

Related Information

The "pthread_attr_getstackaddr Subroutine" on page 909, "pthread_attr_init Subroutine" on page 911, pthread.h file.

Advanced Attributes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_setstacksize Subroutine

Purpose

Sets the value of the stacksize attribute of a thread attributes object.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

```
int pthread attr setstacksize (attr, stacksize)
pthread attr t *attr;
size t stacksize;
```

Description

The pthread_attr_setstacksize subroutine sets the value of the stacksize attribute of the thread attributes object attr. This attribute specifies the minimum stack size, in bytes, of a thread created with this attributes object.

The allocated stack size is always a multiple of 8K bytes, greater or equal to the required minimum stack size of 56K bytes (PTHREAD STACK MIN). The following formula is used to calculate the allocated stack size: if the required stack size is lower than 56K bytes, the allocated stack size is 56K bytes; otherwise, if

the required stack size belongs to the range from (56 + (n - 1) * 16) K bytes to (56 + n * 16) K bytes, the allocated stack size is (56 + n * 16) K bytes.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

attr Specifies the thread attributes object.

stacksize Specifies the minimum stack size, in bytes, to set. The default stack size is

PTHREAD_STACK_MIN. The minimum stack size should be greater or equal than this value.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_attr_setstacksize subroutine is unsuccessful if the following is true:

EINVAL The *attr* parameter is not valid, or the value of the *stacksize* parameter exceeds a system imposed limit.

ENOSYS The stack size POSIX option is not implemented.

Related Information

The **pthread_attr_getstacksize** ("pthread_attr_getstacksize Subroutine" on page 910) subroutine, **pthread_attr_init** ("pthread_attr_init Subroutine" on page 911) subroutine, **pthread_create** ("pthread_create Subroutine" on page 930) subroutine, the **pthread.h** file.

Advanced Attributes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_attr_setsuspendstate_np and pthread_attr_getsuspendstate_np Subroutine

Purpose

Controls whether a thread is created in a suspended state.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread_attr_setsuspendstate_np (attr, suspendstate)
pthread_attr_t *attr;
int suspendstate;
int pthread_attr_getsuspendstate_np (attr, suspendstate)
pthread_attr_t *attr;
int *suspendstate;
```

Description

The suspendstate attribute controls whether the thread is created in a suspended state. If the thread is created suspended, the thread start routine will not execute until pthread_continue_np is run on the thread. The pthread_attr_setsuspendstate_np and pthread_attr_getsuspendstate_np routines, respectively, set and get the *suspendstate* attribute in the *attr* object.

The suspendstate attribute can be set to either PTHREAD_CREATE_SUSPENDED_NP or PTHREAD_CREATE_UNSUSPENDED_NP. A value of PTHREAD_CREATE_SUSPENDED_NP causes all threads created with attr to be in the suspended state, whereas using a value of PTHREAD_CREATE_UNSUSPENDED_NP causes all threads created with attr to be in the unsuspended state. The default value of the suspendstate attribute is PTHREAD CREATE UNSUSPENDED NP.

Parameters

attr Specifies the thread attributes object.

Points to where the *suspendstate* attribute value will be stored. suspendstate

Return Values

Upon successful completion, pthread_attr_setsuspendstate_np and pthread_attr_getsuspendstate_np return a value of 0. Otherwise, an error number is returned to indicate the error.

The pthread attr getsuspendstate np function stores the value of the suspendstate attribute in suspendstate if successful.

Error Codes

The pthread attr setsuspendstate np function will fail if:

EINVAL The value of *suspendstate* is not valid.

pthread_cancel Subroutine

Purpose

Requests the cancellation of a thread.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

int pthread cancel (thread) pthread_t thread;

Description

The pthread_cancel subroutine requests the cancellation of the thread thread. The action depends on the cancelability of the target thread:

- · If its cancelability is disabled, the cancellation request is set pending.
- If its cancelability is deferred, the cancellation request is set pending till the thread reaches a cancellation point.

• If its cancelability is asynchronous, the cancellation request is acted upon immediately; in some cases, it may result in unexpected behavior.

The cancellation of a thread terminates it safely, using the same termination procedure as the **pthread_exit** ("pthread_exit Subroutine" on page 934) subroutine.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

thread Specifies the thread to be canceled.

Return Values

If successful, the **pthread_cancel** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The ptread_cancel function may fail if:

ESRCH No thread could be found corresponding to that specified by the given thread ID.

The pthread_cancel function will not return an error code of EINTR.

Related Information

The **pthread_kill** ("pthread_kill Subroutine" on page 949) subroutine, **pthread_exit** ("pthread_exit Subroutine" on page 934) subroutine, **pthread_join** ("pthread_join or pthread_detach Subroutine" on page 946) subroutine, **pthread_cond_wait** ("pthread_cond_wait or pthread_cond_timedwait Subroutine" on page 925), and **pthread_cond_timedwait** ("pthread_cond_wait or pthread_cond_timedwait Subroutine" on page 925) subroutines.

The pthread.h file.

Terminating Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_cleanup_pop or pthread_cleanup_push Subroutine

Purpose

Activates and deactivates thread cancellation handlers.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

void pthread_cleanup_pop (execute)
int execute;

```
void pthread cleanup push (routine, arg)
void (*routine)(void *);
void *arg;
```

Description

The pthread cleanup push subroutine pushes the specified cancellation cleanup handler routine onto the calling thread's cancellation cleanup stack. The cancellation cleanup handler is popped from the cancellation cleanup stack and invoked with the argument arg when: (a) the thread exits (that is, calls pthread exit, (b) the thread acts upon a cancellation request, or (c) the thread calls pthread cleanup pop with a nonzero execute argument.

The pthread cleanup pop subroutine removes the subroutine at the top of the calling thread's cancellation cleanup stack and optionally invokes it (if execute is nonzero).

These subroutines may be implemented as macros and will appear as statements and in pairs within the same lexical scope (that is, the pthread_cleanup_push macro may be thought to expand to a token list whose first token is '{' with pthread cleanup pop expanding to a token list whose last token is the corresponding '}').

The effect of calling longimp or siglongimp is undefined if there have been any calls to pthread_cleanup_push or pthread_cleanup_pop made without the matching call since the jump buffer was filled. The effect of calling longjmp or siglongjmp from inside a cancellation cleanup handler is also undefined unless the jump buffer was also filled in the cancellation cleanup handler.

Parameters

Specifies if the popped subroutine will be executed. execute Specifies the address of the cancellation subroutine. routine

arg Specifies the argument passed to the cancellation subroutine.

Related Information

The pthread_cancel ("pthread_cancel Subroutine" on page 919), pthread_setcancelstate ("pthread_setcancelstate, pthread_setcanceltype, or pthread_testcancel Subroutines" on page 972) subroutines, the pthread.h file.

Terminating Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread cond destroy or pthread cond init Subroutine

Purpose

Initialize and destroys condition variables.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread cond init (cond, attr)
pthread cond t *cond;
const pthread_condattr_t *attr;
```

```
int pthread cond destroy (cond)
pthread cond t *cond;
pthread_cond_t cond = PTHREAD_COND_INTITIALIZER;
```

Description

The function **pthread cond init** initializes the condition variable referenced by *cond* with attributes referenced by attr. If attr is NULL, the default condition variable attributes are used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable becomes initialized.

Attempting to initialize an already initialized condition variable results in undefined behavior.

The function **pthread_cond_destroy** destroys the given condition variable specified by *cond*; the object becomes, in effect, uninitialized. An implementation may cause pthread cond destroy to set the object referenced by cond to an invalid value. A destroyed condition variable object can be re-initialized using pthread cond init; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro PTHREAD COND INITIALIZER can be used to initialize condition variables that are statically allocated. The effect is equivalent to dynamic initialization by a call to **pthread cond init** with parameter attr specified as NULL, except that no error checks are performed.

Parameters

cond Pointer to the condition variable. attr Specifies the attributes of the condition.

Return Values

If successful, the pthread cond init and pthread cond destroy functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the condition variable specified by cond.

Error Codes

The pthread cond init function will fail if:

EAGAIN The system lacked the necessary resources (other than memory) to initialize another condition variable. **ENOMEM** Insufficient memory exists to initialize the condition variable.

The pthread cond init function may fail if:

EINVAL The value specified by attr is invalid. The **pthread_cond_destroy** function may fail if:

EBUSY The implementation has detected an attempt to destroy the object referenced by cond while it is

referenced (for example, while being used in a pthread cond wait or pthread cond timedwait by

another thread.

EINVAL The value specified by *cond* is invalid.

These functions will not return an error code of EINTR.

Related Information

The pthread cond signal or pthread cond broadcast ("pthread cond signal or pthread_cond_broadcast Subroutine" on page 924) subroutine and the pthread_cond_wait or pthread cond timewait ("pthread cond wait or pthread cond timedwait Subroutine" on page 925) subroutine.

The pthread.h file.

Using Condition Variables in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

PTHREAD COND INITIALIZER Macro

Purpose

Initializes a static condition variable with default attributes.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h> static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

Description

The **PTHREAD COND INITIALIZER** macro initializes the static condition variable *cond*, setting its attributes to default values. This macro should only be used for static condition variables, since no error checking is performed.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Related Information

The pthread cond init ("pthread cond destroy or pthread cond init Subroutine" on page 921) subroutine.

Using Condition Variables in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_cond_signal or pthread_cond_broadcast Subroutine

Purpose

Unblocks one or more threads blocked on a condition.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread cond signal (condition)
pthread_cond_t *condition;
int pthread_cond_broadcast (condition)
pthread_cond_t *condition;
```

Description

These subroutines unblock one or more threads blocked on the condition specified by condition. The pthread_cond_signal subroutine unblocks at least one blocked thread, while the pthread_cond_broadcast subroutine unblocks all the blocked threads.

If more than one thread is blocked on a condition variable, the scheduling policy determines the order in which threads are unblocked. When each thread unblocked as a result of a pthread cond signal or pthread_cond_broadcast returns from its call to pthread_cond_wait or pthread_cond_timedwait, the thread owns the mutex with which it called pthread_cond_waitor pthread_cond_timedwait. The thread(s) that are unblocked contend for the mutex according to the scheduling policy (if applicable), and as if each had called pthread_mutex_lock.

The pthread cond signal or pthread cond broadcast functions may be called by a thread whether or not it currently owns the mutex that threads calling pthread_cond_wait or pthread_cond_timedwait have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex is locked by the thread calling pthread_cond_signal or pthread cond broadcast.

If no thread is blocked on the condition, the subroutine succeeds, but the signalling of the condition is not held. The next thread calling pthread cond wait will be blocked.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

Parameters

condition Specifies the condition to signal.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Code

The pthread_cond_signal and pthread_cond_broadcast subroutines are unsuccessful if the following is true:

EINVAL The *condition* parameter is not valid.

Related Information

The pthread cond wait or pthread cond timedwait ("pthread cond wait or pthread cond timedwait Subroutine") subroutine.

Using Condition Variables in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread cond wait or pthread cond timedwait Subroutine

Purpose

Blocks the calling thread on a condition.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread cond wait (cond, mutex)
pthread_cond_t *cond;
pthread_mutex_t *mutex;
int pthread cond timedwait (cond, mutex, timeout)
pthread cond t *cond;
pthread mutex t *mutex;
const struct timespec *timeout;
```

Description

The pthread cond wait and pthread cond timedwait functions are used to block on a condition variable. They are called with *mutex* locked by the calling thread or undefined behavior will result.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable cond; atomically here means atomically with respect to access by another thread to the mutex and then the condition variable. That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to pthread cond signal or pthread cond broadcast in that thread behaves as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex is locked and owned by the calling thread.

When using condition variables there is always a boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the pthread cond wait or pthread cond timedwait functions may occur. Since the return from pthread_cond_wait or pthread_cond_timedwait does not imply anything about the value of this predicate, the predicate should be reevaluated upon such return.

The effect of using more than one mutex for concurrent pthread cond wait or pthread cond timedwait operations on the same condition variable is undefined; that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding ends when the wait returns.

A condition wait (whether timed or not) is a cancellation point. When the cancelability enable state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancellation request while in a condition wait is that the mutex is (in effect) reacquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to pthread_cond_wait or pthread_cond_timedwait, but at that point notices the cancellation request and instead of returning to the caller of pthread cond wait or pthread cond timedwait, starts the thread cancellation activities, which includes calling cancellation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to pthread cond wait or pthread cond timedwait does not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The pthread cond timedwait function is the same as pthread cond wait except that an error is returned if the absolute time specified by timeout passes (that is, system time equals or exceeds timeout) before the condition cond is signaled or broadcast, or if the absolute time specified by timeout has already been passed at the time of the call. When such time-outs occur, pthread cond timedwait will nonetheless release and reacquire the mutex referenced by mutex. The function pthread cond timedwait is also a cancellation point.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it returns zero due to spurious wakeup.

Parameters

cond Specifies the condition variable to wait on.

Specifies the mutex used to protect the condition variable. The mutex must be locked when the mutex

subroutine is called.

timeout Points to the absolute time structure specifying the blocked state timeout.

Return Values

Except in the case of ETIMEDOUT, all these error checks act as if they were performed immediately at the beginning of processing for the function and cause an error return, in effect, prior to modifying the state of the mutex specified by mutex or the condition variable specified by cond.

Upon successful completion, a value of zero is returned. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_cond_timedwait function will fail if:

The time specified by timeout to pthread_cond_timedwait has passed. **ETIMEDOUT**

The pthread_cond_wait and pthread_cond_timedwait functions may fail if:

EINVAL The value specified by cond, mutex, or timeout is invalid.

EINVAL Different mutexes were supplied for concurrent pthread_cond_wait or pthread_cond_timedwait

operations on the same condition variable.

EINVAL The mutex was not owned by the current thread at the time of the call.

These functions will not return an error code of EINTR.

Related Information

The pthread cond signal orpthread cond broadcast ("pthread cond signal or pthread_cond_broadcast Subroutine" on page 924) subroutine, the pthread.h file.

Using Condition Variables in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_condattr_destroy or pthread_condattr_init Subroutine

Purpose

Initializes and destroys condition variable.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread_condattr_destroy (attr)
pthread_condattr_t *attr;
int pthread condattr init (attr)
pthread condattr t *attr;
```

Description

The function pthread_condattr_init initializes a condition variable attributes object attr with the default value for all of the attributes defined by the implementation. Attempting to initialize an already initialized condition variable attributes object results in undefined behavior.

After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) does not affect any previously initialized condition variables.

The pthread condattr destroy function destroys a condition variable attributes object; the object becomes, in effect, uninitialized. The pthread_condattr_destroy subroutine may set the object referenced by attr to an invalid value. A destroyed condition variable attributes object can be re-initialized using pthread_condattr_init; the results of otherwise referencing the object after it has been destroyed are undefined.

Parameter

attr Specifies the condition attributes object to delete.

Return Values

If successful, the pthread condattr init and pthread condattr destroy functions return zero. Otherwise, an error number is returned to indicate the error.

Error Code

The pthread_condattr_init function will fail if:

ENOMEM Insufficient memory exists to initialize the condition variable attributes object.

The pthread condattr destroy function may fail if:

EINVAL The value specified by attr is invalid.

These functions will not return an error code of EINTR.

Related Information

The pthread cond init ("pthread cond destroy or pthread cond init Subroutine" on page 921) subroutine, pthread condattr getpshared ("pthread condattr getpshared Subroutine") subroutine, pthread create ("pthread create Subroutine" on page 930) subroutine, pthread mutex init ("pthread_mutex_init or pthread_mutex_destroy Subroutine" on page 951) subroutine.

The pthread.h file.

Using Condition Variables in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread condattr getpshared Subroutine

Purpose

Returns the value of the pshared attribute of a condition attributes object.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

int pthread condattr getpshared (attr, pshared) const pthread_condattr_t *attr; int *pshared;

Description

The pthread condattr getpshared subroutine returns the value of the pshared attribute of the condition attribute object attr. This attribute specifies the process sharing of the condition variable created with this attributes object. It may have one of the following values:

PTHREAD PROCESS SHARED Specifies that the condition variable can be used by any thread that has

access to the memory where it is allocated, even if these threads

belong to different processes.

PTHREAD_PROCESS_PRIVATE Specifies that the condition variable shall only be used by threads within

the same process as the thread that created it. This is the default value.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

Parameters

attr Specifies the condition attributes object.

pshared Points to where the pshared attribute value will be stored.

Return Values

Upon successful completion, the value of the pshared attribute is returned via the pshared parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread condattr getpshared subroutine is unsuccessful if the following is true:

EINVAL The attr parameter is not valid.

ENOSYS The process sharing POSIX option is not implemented.

Related Information

The pthread_condattr_setpshared ("pthread_condattr_setpshared Subroutine") subroutine, pthread_condattr_init ("pthread_condattr_destroy or pthread_condattr_init Subroutine" on page 927) subroutine.

Advanced Attributes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_condattr_setpshared Subroutine

Purpose

Sets the value of the pshared attribute of a condition attributes object.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

int pthread_condattr_setpshared (attr, pshared) pthread condattr_t *attr; int pshared;

Description

The pthread_condattr_setpshared subroutine sets the value of the pshared attribute of the condition attributes object attr. This attribute specifies the process sharing of the condition variable created with this attributes object.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

Parameters

attr Specifies the condition attributes object.

pshared Specifies the process sharing to set. It must have one of the following values:

PTHREAD PROCESS SHARED

Specifies that the condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

PTHREAD PROCESS PRIVATE

Specifies that the condition variable shall only be used by threads within the same process as the thread that created it. This is the default value.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread condattr setpshared** subroutine is unsuccessful if the following is true:

EINVAL The attr or pshared parameters are not valid.

Related Information

The pthread condattr getpshared ("pthread condattr getpshared Subroutine" on page 928) subroutine, pthread_condattr_init or pthread_cond_init ("pthread_condattr_destroy or pthread_condattr_init Subroutine" on page 927) subroutine.

Advanced Attributes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread create Subroutine

Purpose

Creates a new thread, initializes its attributes, and makes it runnable.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread_create (thread, attr, start routine (void), arg)
pthread_t *thread;
const pthread attr t *attr;
void **start routine (void);
void *arg;
```

Description

The pthread_create subroutine creates a new thread and initializes its attributes using the thread attributes object specified by the attr parameter. The new thread inherits its creating thread's signal mask; but any pending signal of the creating thread will be cleared for the new thread.

The new thread is made runnable, and will start executing the start routine routine, with the parameter specified by the arg parameter. The arg parameter is a void pointer; it can reference any kind of data. It is not recommended to cast this pointer into a scalar data type (int for example), because the casts may not be portable.

After thread creation, the thread attributes object can be reused to create another thread, or deleted.

The thread terminates in the following cases:

- The thread returned from its starting routine (the main routine for the initial thread)
- The thread called the pthread_exit ("pthread_exit Subroutine" on page 934) subroutine
- · The thread was canceled
- · The thread received a signal that terminated it
- The entire process is terminated due to a call to either the **exec** ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) or **exit** ("exit, atexit, exit, or Exit Subroutine" on page 203) subroutines.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

When multiple threads are created in a process, the FULL_CORE flag is set for all signals. This means that if a core file is produced, it will be much bigger than a single_threaded application. This is necessary to debug multiple-threaded processes.

When a process uses the pthread create function, and thus becomes multi-threaded, the FULL CORE flag is enabled for all signals. If a signal is received whose action is to terminate the process with a core dump, a full dump (usually much larger than a regular dump) will be produced. This is necessary so that multi-threaded programs can be debugged with the **dbx** command.

The following piece of pseudocode is an example of how to avoid getting a full core. Please note that in this case, debug will not be possible. It may be easier to limit the size of the core with the ulimit command.

```
struct sigaction siga;
siga.sa handler = SIG DFL;
siga.sa flags = SA RESTART;
SIGINITSET(siga.as_mask);
sigaction(<SIGNAL NUMBER>, &siga, NULL);
```

The alternate stack is not inherited.

Parameters

thread Points to where the thread ID will be stored.

Specifies the thread attributes object to use in creating the thread. If the value is NULL, the attr

default attributes values will be used.

Points to the routine to be executed by the thread. start_routine

Points to the single argument to be passed to the *start_routine* routine. arg

Return Values

If successful, the pthread_create function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread create function will fail if:

EAGAIN If WLM is running, the limit on the number of threads in the class may have been met.

EINVAL The value specified by attr is invalid.

EPERM The caller does not have appropriate permission to set the required scheduling parameters or

scheduling policy.

The pthread_create function will not return an error code of EINTR.

Related Information

The core file format.

The **dbx** and **ulimit** commands.

The pthread attr init ("pthread attr init Subroutine" on page 911) subroutine, pthread attr destroy ("pthread attr destroy Subroutine" on page 906) subroutine, pthread exit ("pthread exit Subroutine" on page 934) subroutine, pthread cancel ("pthread cancel Subroutine" on page 919) subroutine, pthread kill ("pthread kill Subroutine" on page 949) subroutine, pthread self ("pthread self Subroutine" on page 971) subroutine, **pthread once** ("pthread once Subroutine" on page 961) subroutine, pthread join ("pthread join or pthread detach Subroutine" on page 946) subroutine, fork ("fork, f fork, or vfork Subroutine" on page 248) subroutine, and the pthread.h file.

Creating Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_delay_np Subroutine

Purpose

Causes a thread to wait for a specified period.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

int pthread_delay_np (interval) struct timespec *interval;

Description

The pthread_delay_np subroutine causes the calling thread to delay execution for a specified period of elapsed wall clock time. The period of time the thread waits is at least as long as the number of seconds and nanoseconds specified in the interval parameter.

Notes:

- 1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D THREAD SAFE** compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.
- 2. The **pthread_delay_np** subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters

interval

Points to the time structure specifying the wait period.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread delay np** subroutine is unsuccessful if the following is true:

EINVAL

The interval parameter is not valid.

Related Information

The sleep, nsleep, or usleep subroutine.

pthread_equal Subroutine

Purpose

Compares two thread IDs.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
```

```
int pthread equal (thread1, thread2)
pthread t t\overline{h}read1;
pthread_t thread2;
```

Description

The pthread_equal subroutine compares the thread IDs thread1 and thread2. Since the thread IDs are opaque objects, it should not be assumed that they can be compared using the equality operator (==).

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

thread1 Specifies the first ID to be compared. Specifies the second ID to be compared. thread2

Return Values

The pthread_equal function returns a nonzero value if thread1 and thread2 are equal; otherwise, zero is returned.

If either thread1 or thread2 are not valid thread IDs, the behavior is undefined.

Related Information

The pthread_self ("pthread_self Subroutine" on page 971) subroutine, the pthread_create ("pthread_create Subroutine" on page 930) subroutine, the pthread.h file.

Creating Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread exit Subroutine

Purpose

Terminates the calling thread.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h> void pthread exit (status) void *status;

Description

The pthread_exit subroutine terminates the calling thread safely, and stores a termination status for any thread that may join the calling thread. The termination status is always a void pointer; it can reference any kind of data. It is not recommended to cast this pointer into a scalar data type (int for example), because the casts may not be portable. This subroutine never returns.

Unlike the exit subroutine, the pthread exit subroutine does not close files. Thus any file opened and used only by the calling thread must be closed before calling this subroutine. It is also important to note that the pthread exit subroutine frees any thread-specific data, including the thread's stack. Any data allocated on the stack becomes invalid, since the stack is freed and the corresponding memory may be reused by another thread. Therefore, thread synchronization objects (mutexes and condition variables) allocated on a thread's stack must be destroyed before the thread calls the pthread exit subroutine.

Returning from the initial routine of a thread implicitly calls the pthread_exit subroutine, using the return value as parameter.

If the thread is not detached, its resources, including the thread ID, the termination status, the thread-specific data, and its storage, are all maintained until the thread is detached or the process terminates.

If another thread joins the calling thread, that thread wakes up immediately, and the calling thread is automatically detached.

If the thread is detached, the cleanup routines are popped from their stack and executed. Then the destructor routines from the thread-specific data are executed. Finally, the storage of the thread is reclaimed and its ID is freed for reuse.

Terminating the initial thread by calling this subroutine does not terminate the process, it just terminates the initial thread. However, if all the threads in the process are terminated, the process is terminated by implicitly calling the exit subroutine with a return code of 0 if the last thread is detached, or 1 otherwise. Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

Parameters

status

Points to an optional termination status, used by joining threads. If no termination status is desired, its value should be **NULL**.

Return Values

The pthread exit function cannot return to its caller.

Errors

No errors are defined.

The **pthread exit** function will not return an error code of EINTR.

Related Information

The pthread_cleanup_push ("pthread_cleanup_pop or pthread_cleanup_push Subroutine" on page 920) subroutine, pthread_cleanup_pop ("pthread_cleanup_pop or pthread_cleanup_push Subroutine" on page 920) subroutine, pthread_key_create ("pthread_key_create Subroutine" on page 947) subroutine, pthread_create ("pthread_join ("pthread_join or page 930) subroutine, pthread_join ("pthread_join or pthread_detach Subroutine" on page 946) subroutine, pthread_cancel ("pthread_cancel Subroutine" on page 919) subroutine, exit ("exit, atexit, _exit, or _Exit Subroutine" on page 203) subroutine, the pthread.h file.

Terminating Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread get expiration np Subroutine

Purpose

Obtains a value representing a desired expiration time.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
```

```
int pthread get expiration_np ( delta, abstime)
struct timespec *delta;
struct timespec *abstime;
```

Description

The pthread_get_expiration_np subroutine adds the interval delta to the current absolute system time and returns a new absolute time. This new absolute time can be used as the expiration time in a call to the pthread cond timedwait subroutine.

Notes:

- The pthread.h header file must be the first included file of each source file using the threads library.
 Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
- 2. The **pthread_get_expiration_np** subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters

delta Points to the time structure specifying the interval.

abstime Points to where the new absolute time will be stored.

Return Values

Upon successful completion, the new absolute time is returned via the *abstime* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_get_expiration_np subroutine is unsuccessful if the following is true:

EINVAL The *delta* or *abstime* parameters are not valid.

Related Information

The **pthread_cond_timedwait** ("pthread_cond_wait or pthread_cond_timedwait Subroutine" on page 925) subroutine.

pthread_getconcurrency or pthread_setconcurrency Subroutine

Purpose

Gets or sets level of concurrency.

Library

Threads Library (libthreads.a)

Syntax

```
#include <pthread.h>
int pthread_getconcurrency (void);
int pthread_setconcurrency (new_level)
int new level;
```

Description

The **pthread_setconcurrency** subroutine allows an application to inform the threads implementation of its desired concurrency level, *new_level*. The actual level of concurrency provided by the implementation as a result of this function call is unspecified.

If *new_level* is zero, it causes the implementation to maintain the concurrency level at its discretion as if **pthread_setconcurrency** was never called.

The pthread getconcurrency subroutine returns the value set by a previous call to the pthread setconcurrency subroutine. If the pthread setconcurrency subroutine was not previously called, this function returns zero to indicate that the implementation is maintaining the concurrency level.

When an application calls pthread_setconcurrency, it is informing the implementation of its desired concurrency level. The implementation uses this as a hint, not a requirement.

Use of these subroutines changes the state of the underlying concurrency upon which the application depends. Library developers are advised to not use the pthread_getconcurrency and pthread_setconcurrency subroutines since their use may conflict with an applications use of these functions.

Parameters

new_level

Specifies the value of the concurrency level.

Return Value

If successful, the pthread setconcurrency subroutine returns zero. Otherwise, an error number is returned to indicate the error.

The pthread_getconcurrency subroutine always returns the concurrency level set by a previous call to pthread setconcurrency. If the pthread setconcurrency subroutine has never been called, pthread_getconcurrency returns zero.

Error Codes

The pthread_setconcurrency subroutine will fail if:

EINVAL The value specified by new_level is negative.

EAGAIN The value specific by new_level would cause a system resource to be exceeded.

Related Information

The pthread.h file.

pthread_getrusage_np Subroutine

Purpose

Enable or disable pthread library resource collection, and retrieve resource information for any pthread in the current process.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

int pthread getrusage np (Ptid, RUsage, Mode) pthread_t Ptid; struct rusage *RUsage; int Mode;

Description

The **pthread_getrusage_np** subroutine enables and disables resource collection in the pthread library and collects resource information for any pthread in the current process. When compiled in 64-bit mode, resource usage (rusage) counters are 64-bits for the calling thread. When compiled in 32-bit mode, rusage counters are 32-bits for the calling pthread.

The enable and disable rusage mechanism in the pthread library can also be triggered by an environment variable named **AIXTHREAD_ENRUSG**. Setting this variable to ON enables the functionality, while setting it to OFF disables it. The environment variable can be used along with the

PTHRDSINFO_RUSAGE_START value passed to the **pthread_getrusage_np** subroutine *Mode* parameter. If the environment variable is not to be used, it should be set to NULL.

Parameters

Ptid Specifies the target thread. Must be within the current process.

RUsage

Points to a buffer described in the /usr/include/sys/resource.h file. The fields are defined as follows:

ru_utime

The total amount of time running in user mode.

ru stime

The total amount of time spent in the system executing on behalf of the processes.

ru maxrss

The maximum size, in kilobytes, of the used resident set size.

ru ixrss

An integral value indicating the amount of memory used by the text segment that was also shared among other processes. This value is expressed in *units of kilobytes* X *seconds-of-execution* and is calculated by adding the number of shared memory pages in use each time the internal system clock ticks, and then averaging over one-second intervals.

ru idrss

An integral value of the amount of unshared memory in the data segment of a process, which is expressed in *units of kilobytes* X *seconds-of-execution*.

ru minflt

The number of page faults serviced without any I/O activity. In this case, I/O activity is avoided by reclaiming a page frame from the list of pages awaiting reallocation.

ru majflt

The number of page faults serviced that required I/O activity.

ru nswap

The number of times that a process was swapped out of main memory.

ru_inblock

The number of times that the file system performed input.

ru_oublock

The number of times that the file system performed output.

Note: The numbers that the ru_inblock and ru_oublock fields display account for real I/O only; data supplied by the caching mechanism is charged only to the first process that reads or writes the data.

ru msgsnd

The number of IPC messages sent.

ru_msgrcv

The number of IPC messages received.

ru nsignals

The number of signals delivered.

ru_nvcsw

The number of times a context switch resulted because a process voluntarily gave up the processor before its time slice was completed. This usually occurs while the process waits for a resource to become available.

ru_nivcsw

The number of times a context switch resulted because a higher priority process ran or because the current process exceeded its time slice.

Mode Indicates the task the subroutine should perform. Acceptable values are as follows:

PTHRDSINFO_RUSAGE_START

Enables resource collection in pthread library for all pthreads.

PTHRDSINFO_RUSAGE_STOP

Disables resource collection in pthread library for all pthreads.

PTHRDSINFO RUSAGE COLLECT

Collects resource information for the target thread.

Return Values

Upon successful completion, the **pthread_getrusage_np** subroutine returns a value of 0. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_getrusage_np subroutine fails if:

ENOSYS The calling pthread has given PTHRDSINFO_RUSAGE_COLLECT for *Mode* but the subroutine

was not previously called with PTHRDSINFO_RUSAGE_START for Mode.

EINVAL The address specified for *RUsage* is NULL, not valid, or a null value for *Ptid* was given.

ESRCH Either no thread could be found corresponding to the ID thread of the *Ptid* thread or the thread

corresponding to the Ptid thread ID was not in the current process.

Related Information

The pthreads.h subroutine.

pthread_getschedparam Subroutine

Purpose

Returns the current schedpolicy and schedparam attributes of a thread.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
#include <sys/sched.h>
```

```
int pthread_getschedparam ( thread, schedpolicy, schedparam)
```

pthread_t thread;
int *schedpolicy;

struct sched param *schedparam;

Description

The **pthread_getschedparam** subroutine returns the current schedpolicy and schedparam attributes of the thread *thread*. The schedpolicy attribute specifies the scheduling policy of a thread. It may have one of the following values:

SCHED_FIFO Denotes first-in first-out scheduling. SCHED_RR Denotes round-robin scheduling.

SCHED_OTHER Denotes the default operating system scheduling policy. It is the default value.

The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The sched priority field of the sched param structure contains the priority of the thread. It is an integer value.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

The implementation of this subroutine is dependent on the priority scheduling POSIX option. The priority scheduling POSIX option is implemented in the operating system.

Parameters

thread Specifies the target thread.

schedpolicy Points to where the schedpolicy attribute value will be stored. Points to where the schedparam attribute value will be stored. schedparam

Return Values

Upon successful completion, the current value of the schedpolicy and schedparam attributes are returned via the schedpolicy and schedparam parameters, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_getschedparam** subroutine is unsuccessful if the following is true:

The thread thread does not exist. **ESRCH**

Related Information

The pthread_attr_getschedparam ("pthread_attr_getschedparam Subroutine" on page 908) subroutine.

Threads Scheduling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread getspecific or pthread setspecific Subroutine

Purpose

Returns and sets the thread-specific data associated with the specified key.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
void *pthread_getspecific (key)
pthread_key_t key;
int pthread setspecific (key, value)
pthread key t key;
const void *value;
```

Description

The **pthread_setspecific** function associates a thread-specific *value* with a *key* obtained via a previous call to **pthread_key_create**. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The **pthread_getspecific** function returns the value currently bound to the specified *key* on behalf of the calling thread.

The effect of calling **pthread_setspecific** or **pthread_getspecific** with a *key* value not obtained from **pthread_key_create** or after key has been deleted with **pthread_key_delete** is undefined.

Both **pthread_setspecific** and **pthread_getspecific** may be called from a thread-specific data destructor function. However, calling **pthread_setspecific** from a destructor may result in lost storage or infinite loops.

Parameters

key Specifies the key to which the value is bound.value Specifies the new thread-specific value.

Return Values

The function **pthread_getspecific** returns the thread-specific data value associated with the given key. If no thread-specific data value is associated with key, then the value NULL is returned. If successful, the **pthread_setspecific** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_setspecific function will fail if:

ENOMEM Insufficient memory exists to associate the value with the key.

The pthread_setspecific function may fail if:

EINVAL The key value is invalid.

No errors are returned from pthread_getspecific.

These functions will not return an error code of EINTR.

Related Information

The pthread_key_create ("pthread_key_create Subroutine" on page 947) subroutine, the pthread.h file.

Thread-Specific Data in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread getthrds np Subroutine

Purpose

Retrieves register and stack information for threads.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread getthrds np (thread, mode, buf, bufsize, regbuf, regbufsize)
pthread_t *thread;
int mode;
struct __pthrdsinfo *buf;
int bufsize;
void *regbuf;
int *regbufsize;
```

Description

The **pthread getthrds np** subroutine retrieves information on the state of the thread thread and its underlying kernel thread, including register and stack information.

Parameters

The pointer to the thread. On input it identifies the target thread of the operation, or 0 to operate thread

> on the first entry in the list of threads. On output it identifies the next entry in the list of threads, or 0 if the end of the list has been reached. pthread_getthrds_np can be used to traverse the whole list of threads by starting with thread pointing to 0 and calling pthread_getthrds_np

repeatedly until it returns with thread pointing to 0.

mode Specifies the type of query. These values can be bitwise or'ed together to specify more than one

type of query.

PTHRDSINFO_QUERY_GPRS

get general purpose registers

PTHRDSINFO_QUERY_SPRS

get special purpose registers

PTHRDSINFO_QUERY_FPRS

get floating point registers

PTHRDSINFO_QUERY_REGS

get all of the above registers

PTHRDSINFO_QUERY_TID

get the kernel thread id

PTHRDSINFO QUERY ALL

get everything

buf

Specifies the address of the struct __pthrdsinfo structure that will be filled in by pthread_getthrds_np. On return, this structure holds the following data (depending on the type of query requested):

__pi_ptid

The thread's thread identifier

pi_tid

The thread's kernel thread id, or 0 if the thread does not have a kernel thread

pi_state

The state of the thread, equal to one of the following:

PTHRDSINFO STATE RUN

The thread is running

PTHRDSINFO STATE READY

The thread is ready to run

PTHRDSINFO_STATE_IDLE

The thread is being initialized

PTHRDSINFO STATE SLEEP

The thread is sleeping

PTHRDSINFO STATE TERM

The thread is terminated

PTHRDSINFO_STATE_NOTSUP

Error condition

pi_suspended

1 if the thread is suspended, 0 if it is not

pi_returned

The return status of the thread

__pi_ustk

The thread's user stack pointer

__pi_context

The thread's context (register information)

bufsize

The size of the **pthrdsinfo structure** in bytes.

regbuf

The location of the buffer to hold the register save data from the kernel if the thread is in a

system call.

regbufsize

The pointer to the size of the *regbuf* buffer. On input, it identifies the maximum size of the buffer in bytes. On output, it identifies the number of bytes of register save data. If the thread is not in a system call, there is no register save data returned from the kernel, and regbufsize is 0. If the size of the register save data is larger than the input value of regbufsize, the number of bytes specified by the input value of regbufsize is copied to regbuf, pthread_getthrds_np() returns ERANGE, and the output value of regbufsize specifies the number of bytes required to hold all of

the register save data.

Return Values

If successful, the pthread_getthrds_np function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_getthrds_np function will fail if:

EINVAL Either thread or buf is NULL, or bufsize is not equal to the size of the __pthrdsinfo structure in the

ESRCH No thread could be found corresponding to that specified by the thread ID thread. **ERANGE** regbuf was not large enough to handle all of the register save data.

ENOMEM Insufficient memory exists to perform this operation.

Related Information

The pthread.h file.

pthread_getunique_np Subroutine

Purpose

Returns the sequence number of a thread.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
```

```
int pthread_getunique_np ( thread, sequence)
pthread t *thread;
int *sequence;
```

Description

The pthread_getunique_np subroutine returns the sequence number of the thread thread. The sequence number is a number, unique to each thread, associated with the thread at creation time.

Notes:

- 1. The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
- 2. The pthread_getunique_np subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters

thread Specifies the thread.

sequence Points to where the sequence number will be stored.

Return Values

Upon successful completion, the sequence number is returned via the sequence parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_getunique_np subroutine is unsuccessful if the following is true:

The thread or sequence parameters are not valid. EINVAL

ESRCH The thread thread does not exist.

Related Information

The pthread_self ("pthread_self Subroutine" on page 971) subroutine.

pthread_join or pthread_detach Subroutine

Purpose

Blocks or detaches the calling thread until the specified thread terminates.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>
int pthread_join (thread, status)
pthread_t thread;
void **status;
int pthread_detach (thread)
pthread t thread;

Description

The **pthread_join** subroutine blocks the calling thread until the thread terminates. The target thread's termination status is returned in the *status* parameter.

If the target thread is already terminated, but not yet detached, the subroutine returns immediately. It is impossible to join a detached thread, even if it is not yet terminated. The target thread is automatically detached after all joined threads have been woken up.

This subroutine does not itself cause a thread to be terminated. It acts like the **pthread_cond_wait** subroutine to wait for a special condition.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

The **pthread_detach** subroutine is used to indicate to the implementation that storage for the thread whose thread ID is in the location *thread* can be reclaimed when that thread terminates. This storage shall be reclaimed on process exit, regardless of whether the thread has been detached or not, and may include storage for *thread* return value. If *thread* has not yet terminated, **pthread_detach** shall not cause it to terminate. Multiple **pthread_detach** calls on the same target thread causes an error.

Parameters

thread Specifies the target thread.

status Points to where the termination status of the target thread will be stored. If the value is **NULL**, the

termination status is not returned.

Return Values

If successful, the **pthread_join** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_join and pthread_detach functions will fail if:

EINVAL The implementation has detected that the value specified by thread does not refer to a joinable thread.

ESRCH No thread could be found corresponding to that specified by the given thread ID.

The pthread_join function will fail if:

EDEADLK The value of thread specifies the calling thread.

The **pthread** join function will not return an error code of **EINTR**.

Related Information

The pthread_exit ("pthread_exit Subroutine" on page 934) subroutine, pthread_create ("pthread_create Subroutine" on page 930) subroutine, wait subroutine, pthread cond wait or pthread cond timedwait ("pthread_cond_wait or pthread_cond_timedwait Subroutine" on page 925) subroutines, the pthread.h file.

Joining Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_key_create Subroutine

Purpose

Creates a thread-specific data key.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread key create ( key, destructor )
pthread_key_t * key;
void (* destructor) (void *);
```

Description

The pthread key create subroutine creates a thread-specific data key. The key is shared among all threads within the process, but each thread has specific data associated with the key. The thread-specific data is a void pointer, initially set to NULL.

The application is responsible for ensuring that this subroutine is called only once for each requested key. This can be done, for example, by calling the subroutine before creating other threads, or by using the one-time initialization facility.

Typically, thread-specific data are pointers to dynamically allocated storage. When freeing the storage, the value should be set to NULL. It is not recommended to cast this pointer into scalar data type (int for example), because the casts may not be portable, and because the value of NULL is implementation dependent.

An optional destructor routine can be specified. It will be called for each thread when it is terminated and detached, after the call to the cleanup routines, if the specific value is not NULL. Typically, the destructor routine will release the storage thread-specific data. It will receive the thread-specific data as a parameter. **Note:** The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

key Points to where the key will be stored.

destructor Points to an optional destructor routine, used to cleanup data on thread termination. If no cleanup

is desired, this pointer should be NULL.

Return Values

If successful, the **pthread_key_create** function stores the newly created key value at *key and returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_key_create function will fail if:

EAGAIN The system lacked the necessary resources to create another thread-specific data key, or the

system-imposed limit on the total number of keys per process PTHREAD_KEYS_MAX has been

exceeded.

ENOMEM Insufficient memory exists to create the key.

The pthread key create function will not return an error code of EINTR.

Related Information

The pthread_exit ("pthread_exit Subroutine" on page 934) subroutine, pthread_key_delete ("pthread_key_delete Subroutine") subroutine, pthread_getspecific ("pthread_getspecific or pthread_setspecific Subroutine" on page 941) subroutine, pthread_once ("pthread_once Subroutine" on page 961) subroutine, pthread.h file.

Thread-Specific Data in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_key_delete Subroutine

Purpose

Deletes a thread-specific data key.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

int pthread_key_delete (key)
pthread_key_t key;

Description

The **pthread_key_delete** subroutine deletes the thread-specific data key *key*, previously created with the **pthread_key_create** subroutine. The application must ensure that no thread-specific data is associated with the key. No destructor routine is called.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

Parameters

Specifies the key to delete. key

Return Values

If successful, the pthread key delete function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread key delete function will fail if:

EINVAL The key value is invalid.

The pthread_key_delete function will not return an error code of EINTR.

Related Information

The pthread_key_create ("pthread_key_create Subroutine" on page 947) subroutine, pthread.h file.

Thread-Specific Data in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_kill Subroutine

Purpose

Sends a signal to the specified thread.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <signal.h>
int pthread kill (thread, signal)
pthread t thread;
int signal;
```

Description

The **pthread kill** subroutine sends the signal signal to the thread thread. It acts with threads like the **kill** subroutine with single-threaded processes.

If the receiving thread has blocked delivery of the signal, the signal remains pending on the thread until the thread unblocks delivery of the signal or the action associated with the signal is set to ignore the signal.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

Parameters

thread Specifies the target thread for the signal.

signal Specifies the signal to be delivered. If the signal value is 0, error checking is performed, but no signal is

delivered.

Return Values

Upon successful completion, the function returns a value of zero. Otherwise the function returns an error number. If the **pthread_kill** function fails, no signal is sent.

Error Codes

The pthread_kill function will fail if:

ESRCH No thread could be found corresponding to that specified by the given thread ID. **EINVAL** The value of the *signal* parameter is an invalid or unsupported signal number.

The pthread kill function will not return an error code of EINTR.

Related Information

The **kill** ("kill or killpg Subroutine" on page 478) subroutine, **pthread_cancel** ("pthread_cancel Subroutine" on page 919) subroutine, **pthread_create** ("pthread_create Subroutine" on page 930) subroutine, **sigaction** subroutine, **pthread_self** ("pthread_self Subroutine" on page 971) subroutine, **raise** subroutine, **pthread_h** file.

Signal Management in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_lock_global_np Subroutine

Purpose

Locks the global mutex.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>
void pthread_lock_global_np ()

Description

The **pthread_lock_global_np** subroutine locks the global mutex. If the global mutex is currently held by another thread, the calling thread waits until the global mutex is unlocked. The subroutine returns with the global mutex locked by the calling thread.

Use the global mutex when calling a library package that is not designed to run in a multithreaded environment. (Unless the documentation for a library function specifically states that it is compatible with multithreading, assume that it is not compatible; in other words, assume it is nonreentrant.)

The global mutex is one lock. Any code that calls any function that is not known to be reentrant uses the same lock. This prevents dependencies among threads calling library functions and those functions calling other functions, and so on.

The global mutex is a recursive mutex. A thread that has locked the global mutex can relock it without deadlocking. The thread must then call the pthread unlock global np subroutine as many times as it called this routine to allow another thread to lock the global mutex.

Notes:

- 1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
- 2. The pthread_lock_global_np subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Related Information

The pthread mutex lock ("pthread mutex lock, pthread mutex trylock, or pthread mutex unlock Subroutine" on page 953) subroutine, pthread_unlock_global_np ("pthread_unlock_global_np Subroutine" on page 977) subroutine.

Using Mutexes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_mutex_init or pthread_mutex_destroy Subroutine

Purpose

Initializes or destroys a mutex.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread_mutex_init (mutex, attr)
pthread mutex t *mutex;
const pthread_mutexattr_t *attr;
int pthread mutex destroy (mutex)
pthread_mutex_t *mutex;
```

Description

The pthread mutex init function initializes the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Attempting to initialize an already initialized mutex results in undefined behavior.

The pthread_mutex_destroy function destroys the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized. An implementation may cause pthread_mutex_destroy to set the object referenced by mutex to an invalid value. A destroyed mutex object can be re-initialized using pthread mutex init; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated. The effect is equivalent to dynamic initialization by a call to pthread_mutex_init with parameter attr specified as NULL, except that no error checks are performed.

Parameters

mutex Specifies the mutex to initialize or delete. Specifies the mutex attributes object. attr

Return Values

If successful, the pthread_mutex_init and pthread_mutex_destroy functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks act as if they were performed immediately at the beginning of processing for the function and cause an error return prior to modifying the state of the mutex specified by *mutex*.

Error Codes

The pthread_mutex_init function will fail if:

ENOMEM Insufficient memory exists to initialize the mutex.

EINVAL The value specified by attr is invalid.

The pthread_mutex_destroy function may fail if:

EBUSY The implementation has detected an attempt to destroy the object referenced by mutex while it is locked

or referenced (for example, while being used in a pthread cond waitor pthread cond timedwait by

another thread.

EINVAL The value specified by *mutex* is invalid.

These functions will not return an error code of EINTR.

Related Information

The pthread_mutex_lock, pthread_mutex_trylock ("pthread_mutex_lock, pthread_mutex_trylock, or pthread_mutex_unlock Subroutine" on page 953) subroutine, pthread_mutex_unlock ("pthread mutex lock, pthread mutex trylock, or pthread mutex unlock Subroutine" on page 953) subroutine, pthread mutexattr setpshared ("pthread mutexattr getpshared or pthread mutexattr setpshared Subroutine" on page 960) subroutine.

The pthread.h file.

PTHREAD MUTEX INITIALIZER Macro

Purpose

Initializes a static mutex with default attributes.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Description

The PTHREAD MUTEX INITIALIZER macro initializes the static mutex mutex, setting its attributes to default values. This macro should only be used for static mutexes, as no error checking is performed.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

Related Information

The pthread mutex init ("pthread mutex init or pthread mutex destroy Subroutine" on page 951) subroutine.

Using Mutexes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_mutex_lock, pthread_mutex_trylock, or pthread_mutex_unlock Subroutine

Purpose

Locks and unlocks a mutex.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread mutex lock ( mutex)
pthread_mutex_t *mutex;
int pthread mutex trylock ( mutex)
pthread mutex t *mutex;
int pthread_mutex_unlock ( mutex)
pthread_mutex_t *mutex;
```

Description

The mutex object referenced by the *mutex* parameter is locked by calling **pthread_mutex_lock**. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by the mutex parameter in the locked state with the calling thread as its owner.

If the mutex type is PTHREAD MUTEX NORMAL, deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results.

If the mutex type is PTHREAD MUTEX ERRORCHECK, then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to one. Each time the thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches zero, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

The function pthread_mutex_trylock is identical to pthread_mutex_lock except that if the mutex object referenced by the *mutex* parameter is currently locked (by any thread, including the current thread), the call returns immediately.

The pthread_mutex_unlock function releases the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by the mutex parameter when pthread mutex unlock is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread will acquire the mutex. (In the case of PTHREAD MUTEX RECURSIVE mutexes, the mutex becomes available when the count reaches zero and the calling thread no longer has any locks on this mutex).

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

Parameter

mutex Specifies the mutex to lock.

Return Values

If successful, the pthread mutex lock and pthread mutex unlock functions return zero. Otherwise, an error number is returned to indicate the error.

The function pthread mutex trylock returns zero if a lock on the mutex object referenced by the mutex parameter is acquired. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread mutex trylock function will fail if:

EBUSY The mutex could not be acquired because it was already locked.

The pthread_mutex_lock, pthread_mutex_trylock and pthread_mutex_unlock functions will fail if:

EINVAL The value specified by the *mutex* parameter does not refer to an initialized mutex object.

The **pthread_mutex_lock** function will fail if:

EDEADLK The current thread already owns the mutex and the mutex type is PTHREAD_MUTEX_ERRORCHECK.

The pthread_mutex_unlock function will fail if:

EPERM The current thread does not own the mutex and the mutex type is not PTHREAD_MUTEX_NORMAL.

These functions will not return an error code of EINTR.

Related Information

The pthread_mutex_init or pthread_mutex_destroy ("pthread_mutex_init or pthread_mutex_destroy Subroutine" on page 951) subroutine, pthread.h file.

Using Mutexes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread mutexattr destroy or pthread mutexattr init Subroutine

Purpose

Initializes and destroys mutex attributes.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread mutexattr init (attr)
pthread_mutexattr_t *attr;
int pthread mutexattr destroy (attr)
pthread mutexattr t *attr;
```

Description

The function pthread_mutexattr_init initializes a mutex attributes object attr with the default value for all of the attributes defined by the implementation.

The effect of initializing an already initialized mutex attributes object is undefined.

After a mutex attributes object has been used to initialize one or more mutexes, any function affecting the attributes object (including destruction) does not affect any previously initialized mutexes.

The pthread mutexattr destroy function destroys a mutex attributes object; the object becomes, in effect, uninitialized. An implementation may cause pthread mutexattr destroy to set the object referenced by attr to an invalid value. A destroyed mutex attributes object can be re-initialized using pthread_mutexattr_init; the results of otherwise referencing the object after it has been destroyed are undefined.

Parameters

attr Specifies the mutex attributes object to initialize or delete.

Return Values

Upon successful completion, **pthread_mutexattr_init** and **pthread_mutexattr_destroy** return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_mutexattr_init function will fail if:

ENOMEM Insufficient memory exists to initialize the mutex attributes object.

The pthread_mutexattr_destroy function will fail if:

EINVAL The value specified by *attr* is invalid.

These functions will not return EINTR.

Related Information

The pthread_create ("pthread_create Subroutine" on page 930) subroutine, pthread_mutex_init or pthread_mutex_destroy ("pthread_mutex_init or pthread_mutex_destroy Subroutine" on page 951) subroutine, pthread_cond_destroy or pthread_cond_init ("pthread_cond_destroy or pthread_cond_init Subroutine" on page 921) subroutine, pthread.h file.

Using Mutexes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_mutexattr_getkind_np Subroutine

Purpose

Returns the value of the kind attribute of a mutex attributes object.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

```
int pthread_mutexattr_getkind_np ( attr, kind)
pthread_mutexattr_t *attr;
int *kind;
```

Description

The **pthread_mutexattr_getkind_np** subroutine returns the value of the kind attribute of the mutex attributes object *attr*. This attribute specifies the kind of the mutex created with this attributes object. It may have one of the following values:

MUTEX_FAST_NP

Denotes a fast mutex. A fast mutex can be locked only once. If the same thread unlocks twice the same fast mutex, the thread will deadlock. Any thread can unlock a fast mutex. A fast mutex is not compatible with the priority inheritance protocol.

MUTEX_RECURSIVE_NP Denotes a recursive mutex. A recursive mutex can be locked more than

> once by the same thread without causing that thread to deadlock. The thread must then unlock the mutex as many times as it locked it. Only the thread that locked a recursive mutex can unlock it. A recursive mutex must

not be used with condition variables.

MUTEX_NONRECURSIVE_NP Denotes the default non-recursive POSIX compliant mutex.

Notes:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

2. The pthread_mutexattr_getkind_np subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters

attr Specifies the mutex attributes object.

kind Points to where the kind attribute value will be stored.

Return Values

Upon successful completion, the value of the kind attribute is returned via the kind parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_mutexattr_getkind_np subroutine is unsuccessful if the following is true:

The attr parameter is not valid. **EINVAL**

Related Information

The pthread mutexattr setkind np ("pthread mutexattr setkind np Subroutine" on page 959) subroutine.

Using Mutexes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread mutexattr gettype or pthread mutexattr settype Subroutine

Purpose

Gets or sets a mutex type.

Library

Threads Library (libthreads.a)

Syntax

```
#include <pthread.h>
```

```
int pthread mutexattr gettype (attr, type)
const pthread_mutexattr_t *attr;
int *type;
```

int pthread mutexattr settype (attr, type) pthread mutexattr t *attr; int type;

Description

The pthread mutexattr gettype and pthread mutexattr settype subroutines respectively get and set the mutex type attribute. This attribute is set in the type parameter to these subroutines. The default value of the type attribute is PTHREAD MUTEX DEFAULT. The type of mutex is contained in the type attribute of the mutex attributes. Valid mutex types include:

PTHREAD_MUTEX_NORMAL

This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined

PTHREAD_MUTEX_ERRORCHECK

This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.

PTHREAD_MUTEX_RECURSIVE

A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock which can occur with mutexes of type PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.

PTHREAD_MUTEX_DEFAULT

Attempting to recursively lock a mutex of this type results in undefined behavior. Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behavior. Attempting to unlock a mutex of this type which is not locked results in undefined behavior. An implementation is allowed to map this mutex to one of the other mutex types.

It is advised that an application should not use a PTHREAD MUTEX RECURSIVE mutex with condition variables because the implicit unlock performed for a pthread cond wait or pthread cond timedwait may not actually release the mutex (if it had been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

Parameters

attr Specifies the mutex object to get or set. type Specifies the type to get or set.

Return Values

If successful, the pthread mutexattr settype subroutine returns zero. Otherwise, an error number is returned to indicate the error. Upon successful completion, the pthread mutexattr gettype subroutine returns zero and stores the value of the type attribute of attr into the object referenced by the type parameter. Otherwise an error is returned to indicate the error.

Error Codes

The pthread_mutexattr_gettype and pthread_mutexattr_settype subroutines will fail if:

EINVAL The value of the *type* parameter is invalid.

EINVAL The value specified by the attr parameter is invalid.

Related Information

The pthread cond wait ("pthread cond wait or pthread cond timedwait Subroutine" on page 925) and pthread cond timedwait ("pthread cond wait or pthread cond timedwait Subroutine" on page 925) subroutines.

The pthread.h file.

pthread_mutexattr_setkind_np Subroutine

Purpose

Sets the value of the kind attribute of a mutex attributes object.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread_mutexattr_setkind_np ( attr, kind)
pthread mutexattr t *attr;
int kind;
```

Description

The pthread_mutexattr_setkind_np subroutine sets the value of the kind attribute of the mutex attributes object attr. This attribute specifies the kind of the mutex created with this attributes object.

Notes:

- 1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
- 2. The pthread mutexattr setkind np subroutine is not portable.

This subroutine is provided only for compatibility with the DCE threads. It should not be used when writing new applications.

Parameters

Specifies the mutex attributes object. attr

kind

Specifies the kind to set. It must have one of the following values:

MUTEX_FAST_NP

Denotes a fast mutex. A fast mutex can be locked only once. If the same thread unlocks twice the same fast mutex, the thread will deadlock. Any thread can unlock a fast mutex. A fast mutex is not compatible with the priority inheritance protocol.

MUTEX_RECURSIVE_NP

Denotes a recursive mutex. A recursive mutex can be locked more than once by the same thread without causing that thread to deadlock. The thread must then unlock the mutex as many times as it locked it. Only the thread that locked a recursive mutex can unlock it. A recursive mutex must not be used with condition variables.

MUTEX_NONRECURSIVE_NP

Denotes the default non-recursive POSIX compliant mutex.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_mutexattr_setkind_np subroutine is unsuccessful if the following is true:

EINVAL The *attr* parameter is not valid.

ENOTSUP The value of the *kind* parameter is not supported.

Related Information

The **pthread_mutexattr_getkind_np** ("pthread_mutexattr_getkind_np Subroutine" on page 956) subroutine.

Using Mutexes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_mutexattr_getpshared or pthread_mutexattr_setpshared Subroutine

Purpose

Sets and gets process-shared attribute.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread_mutexattr_getpshared (attr, pshared)
const pthread_mutexattr_t *attr;
int *pshared;
int pthread_mutexattr_setpshared (attr, pshared)
pthread_mutexattr_t *attr;
int pshared;
```

Description

The pthread mutexattr getpshared subroutine obtains the value of the process-shared attribute from the attributes object referenced by attr. The pthread_mutexattr_setpshared subroutine is used to set the process-shared attribute in an initialized attributes object referenced by attr.

The process-shared attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the process-shared attribute is PTHREAD_PROCESS_PRIVATE, the mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, the behavior is undefined. The default value of the attribute is PTHREAD_PROCESS_PRIVATE.

Parameters

Specifies the mutex attributes object. attr

Points to where the pshared attribute value will be stored. pshared

Return Values

Upon successful completion, the pthread_mutexattr_setpshared subroutine returns zero. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the pthread_mutexattr_getpshared subroutine returns zero and stores the value of the process-shared attribute of attr into the object referenced by the pshared parameter. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_mutexattr_getpshared and pthread_mutexattr_setpshared subroutines will fail if:

EINVAL The value specified by attr is invalid.

The pthread_mutexattr_setpshared function will fail if:

EINVAL The new value specified for the attribute is outside the range of legal values for that attribute.

These subroutines will not return an error code of EINTR.

Related Information

The pthread_mutexattr_init ("pthread_mutexattr_destroy or pthread_mutexattr_init Subroutine" on page 955) subroutine.

Advanced Attributes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread once Subroutine

Purpose

Executes a routine exactly once in a process.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread once (once control, init routine)
pthread_once_t *once control;
void (*init_routine)(void);
pthread once_t once_control = PTHREAD_ONCE_INIT;
```

Description

The pthread once subroutine executes the routine init routine exactly once in a process. The first call to this subroutine by any thread in the process executes the given routine, without parameters. Any subsequent call will have no effect.

The *init routine* routine is typically an initialization routine. Multiple initializations can be handled by multiple instances of pthread once t structures. This subroutine is useful when a unique initialization has to be done by one thread among many. It reduces synchronization requirements.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Points to a synchronization control structure. This structure has to be initialized by the static once_control

initializer macro PTHREAD_ONCE_INIT.

init_routine Points to the routine to be executed.

Return Values

Upon successful completion, pthread_once returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

No errors are defined. The pthread_once function will not return an error code of EINTR.

Related Information

The pthread_create ("pthread_create Subroutine" on page 930) subroutine, pthread.h file, PTHREAD ONCE INIT ("PTHREAD ONCE INIT Macro") macro.

One Time Initializations in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

PTHREAD_ONCE_INIT Macro

Purpose

Initializes a once synchronization control structure.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
static pthread_once_t once_block = PTHREAD_ONCE_INIT;
```

Description

The PTHREAD_ONCE_INIT macro initializes the static once synchronization control structure once_block, used for one-time initializations with the pthread_once ("pthread_once Subroutine" on page 961) subroutine. The once synchronization control structure must be static to ensure the unicity of the initialization.

Note: The pthread.h file header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Related Information

The pthread_once ("pthread_once Subroutine" on page 961) subroutine.

One Time Initializations in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread rwlock init or pthread rwlock destroy Subroutine

Purpose

Initializes or destroys a read-write lock object.

Library

Threads Library (libthreads.a)

Syntax

```
#include <pthread.h>
int pthread rwlock init (rwlock, attr)
pthread rwlock t *rwlock;
const pthread_rwlockattr_t *attr;
int pthread rwlock destroy (rwlock)
pthread rwlock t *rwlock;
pthread rwlock t rwlock=PTHREAD RWLOCK INITIALIZER;
```

Description

The pthread rwlock init subroutine initializes the read-write lock referenced by rwlock with the attributes referenced by attr. If attr is NULL, the default read-write lock attributes are used; the effect is the same as passing the address of a default read-write lock attributes object. Once initialized, the lock can be used any number of times without being re-initialized. Upon successful initialization, the state of the read-write lock becomes initialized and unlocked. Results are undefined if pthread rwlock init is called specifying an already initialized read-write lock. Results are undefined if a read-write lock is used without first being initialized.

If the **pthread rwlock init** function fails, *rwlock* is not initialized and the contents of *rwlock* are undefined.

The **pthread_rwlock_destroy** function destroys the read-write lock object referenced by *rwlock* and releases any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is re-initialized by another call to **pthread_rwlock_init**. An implementation may cause **pthread_rwlock_destroy** to set the object referenced by *rwlock* to an invalid value. Results are undefined if **pthread_rwlock_destroy** is called when any thread holds *rwlock*. Attempting to destroy an uninitialized read-write lock results in undefined behavior. A destroyed read-write lock object can be re-initialized using **pthread_rwlock_init**; the results of otherwise referencing the read-write lock object after it has been destroyed are undefined.

In cases where default read-write lock attributes are appropriate, the macro **PTHREAD_RWLOCK_INITIALIZER** can be used to initialize read-write locks that are statically allocated. The effect is equivalent to dynamic initialization by a call to **pthread_rwlock_init** with the parameter *attr* specified as NULL, except that no error checks are performed.

Parameters

rwlock Specifies the read-write lock to be initialized or destroyed.

attr Specifies the attributes of the read-write lock to be initialized.

Return Values

If successful, the **pthread_rwlock_init** and **pthread_rwlock_destroy** functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks, if implemented, will act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the read-write lock specified by *rwlock*.

Error Codes

The pthread rwlock init subroutine will fail if:

ENOMEM Insufficient memory exists to initialize the read-write lock.

EINVAL The value specified by *attr* is invalid.

The pthread_rwlock_destroy subroutine will fail if:

EBUSY The implementation has detected an attempt to destroy the object referenced by rwlock while it is locked.

EINVAL The value specified by *attr* is invalid.

Related Information

The pthread.h file.

The pthread_rwlock_rdlock ("pthread_rwlock_rdlock or pthread_rwlock_tryrdlock Subroutines"), pthread_rwlock_wrlock ("pthread_rwlock_wrlock or pthread_rwlock_trywrlock Subroutines" on page 967), pthread_rwlockattr_init ("pthread_rwlockattr_init or pthread_rwlockattr_destroy Subroutines" on page 969) and pthread_rwlock_unlock ("pthread_rwlock_unlock Subroutine" on page 966) subroutines.

pthread_rwlock_rdlock or pthread_rwlock_tryrdlock Subroutines

Purpose

Locks a read-write lock object for reading.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread rwlock rdlock (rwlock)
pthread_rwlock_t *rwlock;
int pthread rwlock tryrdlock (rwlock)
pthread_rwlock_t *rwlock;
```

Description

The pthread_rwlock_rdlock function applies a read lock to the read-write lock referenced by rwlock. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. It is unspecified whether the calling thread acquires the lock when a writer does not hold the lock and there are writers waiting for the lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the pthread rwlock rdlock call) until it can acquire the lock. Results are undefined if the calling thread holds a write lock on *rwlock* at the time the call is made.

Implementations are allowed to favor writers over readers to avoid writer starvation.

A thread may hold multiple concurrent read locks on rwlock (that is, successfully call the pthread rwlock rdlock function n times). If so, the thread must perform matching unlocks (that is, it must call the **pthread rwlock unlock** function *n* times).

The function pthread rwlock tryrdlock applies a read lock as in the pthread rwlock rdlock function with the exception that the function fails if any thread holds a write lock on rwlock or there are writers blocked on rwlock.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the signal handler the thread resumes waiting for the read-write lock for reading as if it was not interrupted.

Parameters

rwlock

Specifies the read-write lock to be locked for reading.

Return Values

If successful, the pthread rwlock rdlock function returns zero. Otherwise, an error number is returned to indicate the error.

The function pthread_rwlock_tryrdlock returns zero if the lock for reading on the read-write lock object referenced by rwlock is acquired. Otherwise an error number is returned to indicate the error.

Error Codes

The pthread_rwlock_tryrdlock function will fail if:

EBUSY The read-write lock could not be acquired for reading because a writer holds the lock or was blocked on it.

The pthread rwlock rdlock and pthread rwlock tryrdlock functions will fail if:

EINVAL The value specified by rwlock does not refer to an initialized read-write lock object.

EDEADLK The current thread already owns the read-write lock for writing. **EAGAIN**

The read lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded.

Implementation Specifics

Realtime applications may encounter priority inversion when using read-write locks. The problem occurs when a high priority thread 'locks' a read-write lock that is about to be 'unlocked' by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of ways, such as by having critical sections that are guarded by read-write locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

Related Information

The pthread.h file.

The pthread_rwlock_init ("pthread_rwlock_init or pthread_rwlock_destroy Subroutine" on page 963), pthread_rwlock_wrlock ("pthread_rwlock_wrlock or pthread_rwlock_trywrlock Subroutines" on page 967), pthread_rwlockattr_init ("pthread_rwlockattr_init or pthread_rwlockattr_destroy Subroutines" on page 969), and pthread rwlock unlock ("pthread rwlock unlock Subroutine") subroutines.

pthread_rwlock_unlock Subroutine

Purpose

Unlocks a read-write lock object.

Library

Threads Library (libthreads.a)

Syntax

#include <pthread.h>

int pthread_rwlock_unlock (rwlock)
pthread_rwlock_t *rwlock;

Description

The **pthread_rwlock_unlock** subroutine is called to release a lock held on the read-write lock object referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the calling thread.

If this subroutine is called to release a read lock from the read-write lock object and there are other read locks currently held on this read-write lock object, the read-write lock object remains in the read locked state. If this subroutine releases the calling thread's last read lock on this read-write lock object, then the calling thread is no longer one of the owners of the object. If this subroutine releases the last read lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

If this subroutine is called to release a write lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

If the call to the **pthread_rwlock_unlock** subroutine results in the read-write lock object becoming unlocked and there are multiple threads waiting to acquire the read-write lock object for writing, the scheduling policy is used to determine which thread acquires the read-write lock object for writing. If there are multiple threads waiting to acquire the read-write lock object for reading, the scheduling policy is used

to determine the order in which the waiting threads acquire the read-write lock object for reading. If there are multiple threads blocked on rwlock for both read locks and write locks, it is unspecified whether the readers acquire the lock first or whether a writer acquires the lock first.

Results are undefined if any of these subroutines are called with an uninitialized read-write lock.

Parameters

rwlock

Specifies the read-write lock to be unlocked.

Return Values

If successful, the pthread rwlock unlock subroutine returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread rwlock unlock** subroutine may fail if:

The value specified by rwlock does not refer to an initialized read-write lock object. **EINVAL**

EPERM The current thread does not own the read-write lock.

Related Information

The pthread.h file.

The **pthread rwlock init** ("pthread rwlock init or pthread rwlock destroy Subroutine" on page 963), pthread rwlock wrlock ("pthread rwlock wrlock or pthread rwlock trywrlock Subroutines"), pthread_rwlockattr_init ("pthread_rwlockattr_init or pthread_rwlockattr_destroy Subroutines" on page 969), pthread rwlock rdlock ("pthread rwlock rdlock or pthread rwlock tryrdlock Subroutines" on page 964) subroutines.

pthread rwlock wrlock or pthread rwlock trywrlock Subroutines

Purpose

Locks a read-write lock object for writing.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread rwlock wrlock (rwlock)
pthread_rwlock_t *rwlock;
int pthread rwlock trywrlock (rwlock)
pthread rwlock t *rwlock;
```

Description

The pthread_rwlock_wrlock subroutine applies a write lock to the read-write lock referenced by rwlock. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock

rwlock. Otherwise, the thread blocks (that is, does not return from the pthread rwlock wrlock call) until it can acquire the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or write lock) at the time the call is made.

Implementations are allowed to favor writers over readers to avoid writer starvation.

The pthread_rwlock_trywrlock subroutine applies a write lock like the pthread_rwlock_wrlock subroutine, with the exception that the function fails if any thread currently holds rwlock (for reading or writing).

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the signal handler the thread resumes waiting for the read-write lock for writing as if it was not interrupted.

Realtime applications may encounter priority inversion when using read-write locks. The problem occurs when a high priority thread 'locks' a read-write lock that is about to be 'unlocked' by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of ways, such as by having critical sections that are guarded by read-write locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

Parameters

rwlock

Specifies the read-write lock to be locked for writing.

Return Values

If successful, the pthread_rwlock_wrlock subroutine returns zero. Otherwise, an error number is returned to indicate the error.

The pthread rwlock trywrlock subroutine returns zero if the lock for writing on the read-write lock object referenced by rwlock is acquired. Otherwise an error number is returned to indicate the error.

Error Codes

The pthread_rwlock_trywrlock subroutine will fail if:

EBUSY The read-write lock could not be acquired for writing because it was already locked for reading or writing.

The pthread_rwlock_wrlock and pthread_rwlock_trywrlock subroutines may fail if:

EINVAL The value specified by rwlock does not refer to an initialized read-write lock object.

EDEADLK The current thread already owns the read-write lock for writing or reading.

Related Information

The pthread.h file.

The pthread_rwlock_init ("pthread_rwlock_init or pthread_rwlock_destroy Subroutine" on page 963), pthread_rwlock_unlock ("pthread_rwlock_unlock Subroutine" on page 966), pthread_rwlockattr_init ("pthread rwlockattr init or pthread rwlockattr destroy Subroutines"), pthread rwlock rdlock ("pthread rwlock rdlock or pthread rwlock tryrdlock Subroutines" on page 964) subroutines.

pthread_rwlockattr_init or pthread_rwlockattr_destroy Subroutines

Purpose

Initializes and destroys read-write lock attributes object.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread_rwlockattr_init (attr)
pthread_rwlockattr_t *attr;
int pthread rwlockattr destroy (attr)
pthread_rwlockattr_t *attr;
```

Description

The pthread_rwlockattr_init subroutine initializes a read-write lock attributes object attr with the default value for all of the attributes defined by the implementation. Results are undefined if pthread rwlockattr init is called specifying an already initialized read-write lock attributes object.

After a read-write lock attributes object has been used to initialize one or more read-write locks, any function affecting the attributes object (including destruction) does not affect any previously initialized read-write locks.

The pthread_rwlockattr_destroy subroutine destroys a read-write lock attributes object. The effect of subsequent use of the object is undefined until the object is re-initialized by another call to pthread_rwlockattr_init. An implementation may cause pthread_rwlockattr_destroy to set the object referenced by attr to an invalid value.

Parameters

attr

Specifies a read-write lock attributes object to be initialized or destroyed.

Return Value

If successful, the pthread_rwlockattr_init and pthread_rwlockattr_destroy subroutines return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_rwlockattr_init subroutine will fail if:

ENOMEM Insufficient memory exists to initialize the read-write lock attributes object.

The pthread_rwlockattr_destroy subroutine will fail if:

EINVAL The value specified by attr is invalid.

Related Information

The pthread.h file.

The pthread_rwlock_init ("pthread_rwlock_init or pthread_rwlock_destroy Subroutine" on page 963), pthread_rwlock_unlock ("pthread_rwlock_unlock Subroutine" on page 966), pthread_rwlock_wrlock ("pthread_rwlock_wrlock or pthread_rwlock_trywrlock Subroutines" on page 967), pthread_rwlock_rdlock ("pthread_rwlock_rdlock Subroutines" on page 964), and pthread_rwlockattr_getpshared ("pthread_rwlockattr_getpshared or pthread_rwlockattr_setpshared Subroutines") subroutines.

pthread_rwlockattr_getpshared or pthread_rwlockattr_setpshared Subroutines

Purpose

Gets and sets process-shared attribute of read-write lock attributes object.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>
int pthread_rwlockattr_getpshared (attr, pshared)
const pthread_rwlockattr_t *attr;
int *pshared;
int pthread_rwlockattr_setpshared (attr, pshared)
pthread_rwlockattr_t *attr;
int pshared;

Description

The process-shared attribute is set to PTHREAD_PROCESS_SHARED to permit a read-write lock to be operated upon by any thread that has access to the memory where the read-write lock is allocated, even if the read-write lock is allocated in memory that is shared by multiple processes. If the process-shared attribute is PTHREAD_PROCESS_PRIVATE, the read-write lock will only be operated upon by threads created within the same process as the thread that initialized the read-write lock; if threads of differing processes attempt to operate on such a read-write lock, the behavior is undefined. The default value of the process-shared attribute is PTHREAD_PROCESS_PRIVATE.

The **pthread_rwlockattr_getpshared** subroutine obtains the value of the process-shared attribute from the initialized attributes object referenced by *attr*. The **pthread_rwlockattr_setpshared** subroutine is used to set the process-shared attribute in an initialized attributes object referenced by *attr*.

Parameters

attr Specifies the initialized attributes object.

pshared Specifies the process-shared attribute of read-write lock attributes object to be gotten

and set.

Return Values

If successful, the **pthread_rwlockattr_setpshared** subroutine returns zero. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the pthread_rwlockattr_getpshared subroutine returns zero and stores the value of the process-shared attribute of attr into the object referenced by the pshared parameter. Otherwise an error number is returned to indicate the error.

Error Codes

The pthread_rwlockattr_getpshared and pthread_rwlockattr_setpshared subroutines will fail if:

EINVAL The value specified by attr is invalid.

The pthread rwlockattr setpshared subroutine will fail if:

EINVAL The new value specified for the attribute is outside the range of legal values for that attribute.

Related Information

The pthread.h file.

The **pthread rwlock init** ("pthread rwlock init or pthread rwlock destroy Subroutine" on page 963), pthread rwlock unlock ("pthread rwlock unlock Subroutine" on page 966), pthread rwlock wrlock ("pthread_rwlock_wrlock or pthread_rwlock_trywrlock Subroutines" on page 967), pthread_rwlock_rdlock ("pthread_rwlock_rdlock or pthread_rwlock_tryrdlock Subroutines" on page 964), pthread_rwlockattr_init ("pthread_rwlockattr_init or pthread_rwlockattr_destroy Subroutines" on page 969) subroutines.

pthread self Subroutine

Purpose

Returns the calling thread's ID.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h> pthread t pthread self (void);

Description

The **pthread_self** subroutine returns the calling thread's ID.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

Return Values

The calling thread's ID is returned.

Errors

No errors are defined.

The pthread_self function will not return an error code of EINTR.

Related Information

The pthread_create ("pthread_create Subroutine" on page 930) subroutine, pthread_equal ("pthread_equal Subroutine" on page 933) subroutine.

The pthread.h file.

Creating Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread setcancelstate, pthread setcanceltype, or pthread testcancel **Subroutines**

Purpose

Sets the calling thread's cancelability state.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread setcancelstate (state, oldstate)
int state;
int *oldstate;
int pthread_setcanceltype (type, oldtype)
int type:
int *oldtype;
int pthread_testcancel (void)
```

Description

The pthread setcancelstate subroutine atomically both sets the calling thread's cancelability state to the indicated state and returns the previous cancelability state at the location referenced by oldstate. Legal values for state are PTHREAD CANCEL ENABLE and PTHREAD CANCEL DISABLE.

The pthread_setcanceltype subroutine atomically both sets the calling thread's cancelability type to the indicated type and returns the previous cancelability type at the location referenced by oldtype. Legal values for type are PTHREAD CANCEL DEFERRED and PTHREAD CANCEL ASYNCHRONOUS.

The cancelability state and type of any newly created threads, including the thread in which main was first invoked, are PTHREAD CANCEL ENABLE and PTHREAD CANCEL DEFERRED respectively.

The **pthread testcancel** subroutine creates a cancellation point in the calling thread. The pthread testcancel subroutine has no effect if cancelability is disabled.

Parameters

state Specifies the new cancelability state to set. It must have one of the following values:

PTHREAD_CANCEL_DISABLE

Disables cancelability; the thread is not cancelable. Cancellation requests are held pending.

PTHREAD CANCEL ENABLE

Enables cancelability; the thread is cancelable, according to its cancelability type. This is the

default value.

oldstate Points to where the previous cancelability state value will be stored.

type Specifies the new cancelability type to set.

oldtype Points to where the previous cancelability type value will be stored.

Return Values

If successful, the **pthread_setcanceIstate** and **pthread_setcanceItype** subroutines return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The pthread_setcancelstate subroutine will fail if:

EINVAL The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

The pthread_setcanceltype subroutine will fail if:

EINVAL The specified type is not PTHREAD_CANCEL_DEFERRED or PTHREAD_CANCEL_ASYNCHRONOUS.

These subroutines will not return an error code of EINTR.

Related Information

The **pthread_cancel** ("pthread_cancel Subroutine" on page 919) subroutine.

The pthread.h file.

Terminating Threads in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_setschedparam Subroutine

Purpose

Returns the current schedpolicy and schedparam attributes of a thread.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>
#include <sys/sched.h>

int pthread_setschedparam (thread, schedpolicy, schedparam)
pthread_t thread;
int schedpolicy;
const struct sched param *schedparam;

Description

The pthread_setschedparam subroutine dynamically sets the schedpolicy and schedparam attributes of the thread thread. The schedpolicy attribute specifies the scheduling policy of the thread. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The sched priority field of the sched_param structure contains the priority of the thread. It is an integer value.

If the target thread has system contention scope, the process must have root authority to set the scheduling policy to either SCHED_FIFO or SCHED_RR.

Note: The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

This subroutine is part of the Base Operating System (BOS) Runtime. The implementation of this subroutine is dependent on the priority scheduling POSIX option. The priority scheduling POSIX option is implemented in the operating system.

Parameters

thread Specifies the target thread.

Points to the schedpolicy attribute to set. It must have one of the following values: schedpolicy

SCHED FIFO

Denotes first-in first-out scheduling.

SCHED RR

Denotes round-robin scheduling.

SCHED OTHER

Denotes the default operating system scheduling policy. It is the default value.

Note: It is not permitted to change the priority of a thread when setting its scheduling policy to SCHED_OTHER. In this case, the priority is managed directly by the kernel, and the only legal value that can be passed to pthread_setschedparam is DEFAULT_PRIO, which is defined in

pthread.h as 1.

Points to where the scheduling parameters to set are stored. The sched_priority field must be schedparam

in the range from 1 to 127, where 1 is the least favored priority, and 127 the most favored.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread setschedparam subroutine is unsuccessful if the following is true:

EINVAL The thread or schedparam parameters are not valid. **ENOSYS** The priority scheduling POSIX option is not implemented.

ENOTSUP The value of the schedpolicy or schedparam attributes are not supported.

EPERM The target thread has insufficient permission to perform the operation or is already engaged in a mutex

protocol.

ESRCH The thread thread does not exist.

Related Information

The pthread getschedparam ("pthread getschedparam Subroutine" on page 940) subroutine, pthread attr setschedparam ("pthread attr setschedparam Subroutine" on page 915) subroutine. Threads Scheduling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread_sigmask Subroutine

Purpose

Examines and changes blocked signals.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <signal.h>
int pthread_sigmask (how, set, oset)
int how;
const sigset t *set;
sigset_t *oset;
```

Description

Refer to sigthreadmask in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

pthread signal to cancel np Subroutine

Purpose

Cancels the specified thread.

Library

Threads Library (libpthreads.a)

Syntax

```
#include <pthread.h>
int pthread_signal_to_cancel_np ( sigset, thread)
sigset t *sigset;
pthread t *thread;
```

Description

The pthread_signal_to_cancel_np subroutine cancels the target thread thread by creating a handler thread. The handler thread calls the sigwait subroutine with the sigset parameter, and cancels the target thread when the sigwait subroutine returns. Successive calls to this subroutine override the previous ones.

Notes:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D THREAD SAFE** compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

2. The pthread signal to cancel np subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters

sigset Specifies the set of signals to wait on.

thread Specifies the thread to cancel.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The pthread_signal_to_cancel_np subroutine is unsuccessful if the following is true:

EAGAIN The handler thread cannot be created.

EINVAL The *sigset* or *thread* parameters are not valid.

Related Information

The pthread_cancel ("pthread_cancel Subroutine" on page 919) subroutine, sigwait subroutine.

pthread_suspend_np and pthread_continue_np Subroutine

Purpose

Suspends execution of the pthread specified by thread.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h>

int pthread_suspend_np(thread)
pthread t thread;

int pthread_continue_np(thread)
pthread t thread;

Description

The pthread_suspend_np subroutine immediately suspends the execution of the pthread specified by thread. On successful return from pthread_suspend_np, the suspended pthread is no longer executing. If pthread_suspend_np is called for a pthread that is already suspended, the pthread is unchanged and pthread suspend np returns successful.

The **pthread_continue_np** routine resumes the execution of a suspended pthread. If **pthread_continue_np** is called for a pthread that is not suspended, the pthread is unchanged and **pthread_continue_np** returns successful.

A suspended pthread will not be awakened by a signal. The signal stays pending until the execution of pthread is resumed by **pthread_continue_np**.

Note: Using pthread suspend np should only be used by advanced users because improper use of this subcommand can lead to application deadlock or the target thread may be suspended holding application locks. Nested suspension is not supported: a pthread that is suspended twice is continued by a single pthread continue.

Parameters

thread

Specifies the target thread.

Return Values

Zero is returned when successful. A nonzero value indicates an error.

Error Codes

If any of the following conditions occur, pthread_suspend_np and pthread_continue_np fail and return the corresponding value:

ESRCH

The thread attribute cannot be found in the current process.

pthread unlock global np Subroutine

Purpose

Unlocks the global mutex.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h> void pthread unlock global np ()

Description

The pthread unlock global np subroutine unlocks the global mutex when each call to the pthread lock global np subroutine is matched by a call to this routine. For example, if a thread called the pthread lock global np three times, the global mutex is unlocked after the third call to the pthread_unlock_global_np subroutine.

If no threads are waiting for the global mutex, it becomes unlocked with no current owner. If one or more threads are waiting to lock the global mutex, exactly one thread returns from its call to the pthread_lock_global_np subroutine.

Notes:

- 1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the -D_THREAD_SAFE compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
- 2. The pthread_unlock_global_np subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Related Information

The pthread_lock_global_np ("pthread_lock_global_np Subroutine" on page 950) subroutine.

Using Mutexes in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

pthread yield Subroutine

Purpose

Forces the calling thread to relinquish use of its processor.

Library

Threads Library (libpthreads.a)

Syntax

#include <pthread.h> void pthread_yield ()

Description

The pthread_yield subroutine forces the calling thread to relinquish use of its processor, and to wait in the run queue before it is scheduled again. If the run queue is empty when the pthread_yield subroutine is called, the calling thread is immediately rescheduled.

If the thread has global contention scope (PTHREAD SCOPE SYSTEM), calling this subroutine acts like calling the yield subroutine. Otherwise, another local contention scope thread is scheduled.

The pthread.h header file must be the first included file of each source file using the threads library. Otherwise, the -D THREAD SAFE compilation flag should be used, or the cc r compiler used. In this case, the flag is automatically set.

Related Information

The **yield** subroutine and the **sched yield** subroutine.

Threads Scheduling in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Threads Library Options in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

ptrace, ptracex, ptrace64 Subroutine

Purpose

Traces the execution of another process.

Library

Standard C Library (libc.a)

Syntax

#include <sys/reg.h> #include <sys/ptrace.h> #include <sys/ldr.h>

```
int ptrace ( Request, Identifier, Address, Data, Buffer)
int Request;
int Identifier;
int *Address;
int Data;
int *Buffer;
int ptracex ( request, identifier, addr, data, buff)
int request;
int identifier;
long long addr;
int data;
int *buff;
int ptrace64 ( request, identifier, addr, data, buff)
int request;
long long identifier;
long long addr;
int data;
int *buff;
```

Description

The ptrace subroutine allows a 32-bit process to trace the execution of another process. The ptrace subroutine is used to implement breakpoint debugging.

A debugged process executes normally until it encounters a signal. Then it enters a stopped state and its debugging process is notified with the wait subroutine.

Exception: If the process encounters the SIGTRAP signal, a signal handler for SIGTRAP exists, and fast traps ("Fast Trap Instructions" on page 980) have been enabled for the process, then the signal handler is called and the debugger is not notified. This exception only applies to AIX 4.3.3 and later releases.

While the process is in the stopped state, the debugger examines and modifies the memory image of the process being debugged by using the ptrace subroutine. For multi-threaded processes, the getthrds ("getthrds Subroutine" on page 372) subroutine identifies each kernel thread in the debugged process. Also, the debugging process can cause the debugged process to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

As a security measure, the ptrace subroutine inhibits the set-user-ID facility on subsequent exec subroutines.

(This paragraph only applies to AIX 4.3.1 and previous releases.) When a process is executing under ptrace control, portions of the process's address space are recopied after load, unload, and loadbind calls. For a 32-bit process, the main program text (loaded in segment 1) and shared library modules (loaded in segment 13) is recopied. Any breakpoints or other modifications to these segments must be reinserted after load, unload, or loadbind. Changes to privately loaded modules persist. For a 64-bit process, shared library modules are recopied after load and unload are called. (For AIX 4.3 and AIX 4.3.1, these segments have a virtual address of 0x09000000xxxxxxxx, where x denotes any value.) The segments for the main programs and the segments containing privately loaded modules are not recopied. When a 64-bit process calls loadbind, no segments are recopied and the debugger is not notified.

(This paragraph only applies to AIX 4.3.2 and later releases.) When a process executing under ptrace control calls load or unload, the debugger is notified and the W SLWTED flag is set in the status returned by wait. (A 32-bit process calling loadbind is stopped as well.) If the process being debugged has added

modules in the shared library to its address space, the modules are added to the process's private copy of the shared library segments. If shared library modules are removed from a process's address space, the modules are deleted from the process's private copy of the library text segment by freeing the pages that contain the module. No other changes to the segment are made, and existing breakpoints do not have to be reinserted.

To allow a debugger to generate code more easily (in order to handle fast trap instructions, for example), memory from the end of the main program up to the next segment boundary can be modified. That memory is read-only to the process but can be modified by the debugger.

When a process being traced forks, the child process is initialized with the unmodified main program and shared library segment, effectively removing breakpoints in these segments in the child process. If multiprocess debugging is enabled, new copies of the main program and shared library segments are made. Modifications to privately loaded modules, however, are not affected by a fork. These breakpoints will remain in the child process, and if these breakpoints are executed, a SIGTRAP signal will be generated and delivered to the process.

If a traced process initiates an exec subroutine, the process stops before executing the first instruction of the new image and returns the SIGTRAP signal.

Note: ptrace and **ptracex** are not supported in 64-bit mode.

Fast Trap Instructions

Note: The "Fast Trap Instructions" section only applies to AIX 4.3.3 and later releases.

Sometimes, allowing the process being debugged to handle certain trap instructions is useful, instead of causing the process to stop and notify the debugger. You can use this capability to patch running programs or programs whose source codes are not available. For a process to use this capability, you must enable fast traps, which requires you to make a ptrace call from a debugger on behalf of the process.

To let a process handle fast traps, a debugger uses the ptrace (PT_SET, pid, 0, PTFLAG_FAST_TRAP, 0) subroutine call. Cancel this capability with the ptrace (PT_CLEAR, pid, 0, PTFLAG_FAST_TRAP, 0) subroutine call. If a process is able to handle fast traps when the debugger detaches, the fast trap capability remains in effect. Consequently, when another debugger attaches to that process, fast trap processing is still enabled. When no debugger is attached to a process, SIGTRAP signals are handled in the same manner, regardless of whether fast traps are enabled.

A fast trap instruction is an unconditional trap immediate instruction in the form twi 14,r13,0xNXXX. This instruction has the binary form 0x0ddfNXXX, where N is a hex digit >=8 and XXX are any three hex digits. By using different values of 0xNXXX, a debugger can generate different fast trap instructions, allowing a signal handler to quickly determine how to handle the signal. (The fast trap instruction is defined by the macro _PTRACE_FASTTRAP. The _PTRACE_FASTTRAP_MASK macro can be used to check whether a trap is a fast trap.)

Usually, a fast trap instruction is treated like any other trap instruction. However, if a process has a signal handler for SIGTRAP, the signal is not blocked, and the fast trap capability is enabled, then the signal handler is called and the debugger is not notified.

A signal handler can logically AND the trap instruction with **PTRACE FASTTRAP NUM** (0x7FFF) to obtain an integer identifying which trap instruction was executed.

For the 64-bit Process

Use **ptracex** where the debuggee is a 64-bit process and the operation requested uses the third (Address) parameter to reference the debuggee's address space or is sensitive to register size. Use ptrace64 if the second parameter (identifier) is a thread id from a 64-bit process. Note that ptracex and ptrace64 will also support 32-bit debugees.

If returning or passing an int doesn't work for a 64-bit debuggee (for example, PT READ GPR), the buffer parameter takes the address for the result. Thus, with the ptracex subroutine, PT READ GPR and PT WRITE GPR take a pointer to an 8 byte area representing the register value.

In general, ptracex supports all the calls that ptrace does when they are modified for any that are extended for 64-bit addresses (for example, GPRs, LR, CTR, IAR, and MSR). Anything whose size increases for 64-bit processes must be allowed for in the obvious way (for example, PT_REGSET must be an array of long longs for a 64-bit debuggee).

Parameters

Reauest

Determines the action to be taken by the **ptrace** subroutine and has one of the following values:

PT ATTACH

This request allows a debugging process to attach a current process and place it into trace mode for debugging. This request cannot be used if the target process is already being traced. The Identifier parameter is interpreted as the process ID of the traced process. The Address, Data, and Buffer parameters are ignored.

If this request is unsuccessful, -1 is returned and the errno global variable is set to one the following codes:

ESRCH

Process ID is not valid; the traced process is a kernel process; the process is currently being traced; or, the debugger or traced process already exists.

EPERM

Real or effective user ID of the debugger does not match that of the traced process, or the debugger does not have root authority.

EINVAL

The debugger and the traced process are the same.

PT CLEAR

This request clears an internal flag or capability. The Data parameter specifies which flags to clear. The following flag can be cleared:

PTFLAG_FAST_TRAP

Disables the special handling of a fast trap instruction ("Fast Trap Instructions" on page 980). This allows all fast trap instructions causing an interrupt to generate a **SIGTRAP** signal.

The Identifier parameter specifies the process ID of the traced process. The Address parameter, Buffer parameter, and the unused bits in the Data parameter are reserved for future use and should be set to 0.

PT CONTINUE

This request allows the process to resume execution. If the Data parameter is 0, all pending signals, including the one that caused the process to stop, are concealed before the process resumes execution. If the Data parameter is a valid signal number, the

process resumes execution as if it had received that signal. If the Address parameter equals 1, the execution continues from where it stopped. If the Address parameter is not 1, it is assumed to be the address at which the process should resume execution. Upon successful completion, the value of the Data parameter is returned to the debugging process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, -1 is returned and the errno global variable is set to the following code:

EIO The signal to be sent to the traced process is not a valid signal number.

Note: For the PT_CONTINUE request, use ptracex or prtrace64 with a 64-bit debuggee because the resume address needs 64 bits.

PTT_CONTINUE

This request asks the scheduler to resume execution of the kernel thread specified by Identifier. This kernel thread must be the one that caused the exception. The Data parameter specifies how to handle signals:

- If the Data parameter is 0, the kernel thread which caused the exception will be resumed as if the signal never occurred.
- If the Data parameter is a valid signal number, the kernel thread which caused the exception will be resumed as if it had received that signal.

The *Address* parameter specifies where to resume execution:

- If the Address parameter is 1, execution resumes from the address where it stopped.
- If the Address parameter contains an address value other than 1, execution resumes from that address.

The Buffer parameter should point to a PTTHREADS structure, which contains a list of kernel thread identifiers to be started. This list should be NULL terminated if it is smaller than the maximum allowed.

On successful completion, the value of the Data parameter is returned to the debugging process. On unsuccessful completion, the value -1 is returned, and the errno global variable is set as follows:

EINVAL

The *Identifier* parameter names the wrong kernel thread.

EIO The signal to be sent to the traced kernel thread is not a valid signal number.

ESRCH

The Buffer parameter names an invalid kernel thread. Each kernel thread in the list must be stopped and belong to the same process as the kernel thread named by the *Identifier* parameter.

Note: For the PTT CONTINUE request, use ptracex or ptrace64 with a 64-bit debuggee because the resume address needs 64 bits.

PT DETACH

This request allows a debugged process, specified by the *Identifier* parameter, to exit trace mode. The process then continues running, as if it had received the signal whose number is contained in the Data parameter. The process is no longer traced and does not process any further ptrace calls. The Address and Buffer parameters are ignored.

If this request is unsuccessful, -1 is returned and the errno global variable is set to the following code:

EIO Signal to be sent to the traced process is not a valid signal number.

PT KILL

This request allows the process to terminate the same way it would with an exit subroutine.

PT_LDINFO

This request retrieves a description of the object modules that were loaded by the debugged process. The Identifier parameter is interpreted as the process ID of the traced process. The Buffer parameter is ignored. The Address parameter specifies the location where the loader information is copied. The Data parameter specifies the size of this area. The loader information is retrieved as a linked list of Id_info structures. The first element of the list corresponds to the main executable module. The Id info structures are defined in the /usr/include/sys/ldr.h file. The linked list is implemented so that the 1dinfo nxt field of each element gives the offset of the next element from this element. The 1dinfo nxt field of the last element has the value 0.

Each object module reported is opened on behalf of the debugger process. The file descriptor and file pointer for an object module are recorded in the 1dinfo fd and ldinfo fp fields of the corresponding Id info structure, respectively. The debugger process is responsible for managing the files opened by the **ptrace** subroutine.

If this request is unsuccessful, -1 is returned and the errno global variable is set to the following code:

ENOMEM

Either the area is not large enough to accommodate the loader information, or there is not enough memory to allocate an equivalent buffer in the kernel.

Note: For the PT_LDINFO request, use ptracex or ptrace64 with a 64-bit debuggee because the source address needs 64 bits.

PT MULTI

This request turns multiprocess debugging mode on and off, to allow debugging to continue across fork and exec subroutines. A 0 value for the Data parameter turns multiprocess debugging mode off, while all other values turn it on. When multiprocess debugging mode is in effect, any fork subroutine allows both the traced process and its newly created process to trap on the next instruction. If a traced process initiated an exec subroutine, the process stops before executing the first instruction of the new image and returns the SIGTRAP signal. The Identifier parameter is interpreted as the process ID of the traced process. The Address and Buffer parameters are ignored.

Also, when multiprocess debugging mode is enabled, the following values are returned from the wait subroutine:

W SEWTED

Process stopped during execution of the exec subroutine.

W SFWTED

Process stopped during execution of the **fork** subroutine.

PT READ BLOCK

This request reads a block of data from the debugged process address space. The Address parameter points to the block of data in the process address space, and the Data parameter gives its length in bytes. The value of the Data parameter must not be greater than 1024. The *Identifier* parameter is interpreted as the process ID of the traced process. The Buffer parameter points to the location in the debugging process address space where the data is copied. Upon successful completion, the ptrace subroutine returns the value of the Data parameter.

If this request is unsuccessful, -1 is returned and the errno global variable is set to one of the following codes:

EIO The Data parameter is less than 1 or greater than 1024.

EIO The Address parameter is not a valid pointer into the debugged process address space.

EFAULT

The Buffer parameter does not point to a writable location in the debugging process address space.

Note: For the PT READ BLOCK request, use ptracex or ptrace64 with a 64-bit debuggee because the source address needs 64 bits.

PT READ FPR

This request stores the value of a floating-point register into the location pointed to by the Address parameter. The Data parameter specifies the floating-point register, defined in the sys/reg.h file for the machine type on which the process is executed. The Identifier parameter is interpreted as the process ID of the traced process. The Buffer parameter is ignored.

If this request is unsuccessful, -1 is returned and the errno global variable is set to the following code:

EIO The Data parameter is not a valid floating-point register. The Data parameter must be in the range 256-287.

PTT READ FPRS

This request writes the contents of the 32 floating point registers to the area specified by the Address parameter. This area must be at least 256 bytes long. The Identifier parameter specifies the traced kernel thread. The Data and Buffer parameters are ignored.

PT READ GPR

This request returns the contents of one of the general-purpose or special-purpose registers of the debugged process. The Address parameter specifies the register whose value is returned. The value of the *Address* parameter is defined in the **sys/req.h** file for the machine type on which the process is executed. The Identifier parameter is interpreted as the process ID of the traced process. The Data and Buffer parameters are ignored. The buffer points to long long target area.

Note: If ptracex or ptrace64 with a 64-bit debuggee is used for this request, the register value is instead returned to the 8-byte area pointed to by the buffer pointer.

If this request is unsuccessful, -1 is returned and the errno global variable is set to the following code:

EIO The Address is not a valid general-purpose or special-purpose register. The Address parameter must be in the range 0-31 or 128-136.

PTT READ GPRS

This request writes the contents of the 32 general purpose registers to the area specified by the Address parameter. This area must be at least 128 bytes long.

Note: If ptracex or ptrace64 are used with a 64-bit debuggee for the PTT_READ_GPRS request, there must be at least a 256 byte target area. The *Identifier* parameter specifies the traced kernel thread. The Data and Buffer parameters are ignored.

PT_READ_I or PT_READ_D

These requests return the word-aligned address in the debugged process address space specified by the Address parameter. On all machines currently supported by AIX Version 4, the PT READ I and PT READ D instruction and data requests can be used with equal results. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Data* parameter is ignored.

If this request is unsuccessful, -1 is returned and the errno global variable is set to the following code:

EIO The Address is not word-aligned, or the Address is not valid. User blocks, kernel segments, and kernel extension segments are not considered as valid addresses.

Note: For the PT_READ_I or the PT_READ_D request, use ptracex or ptrace64 with a 64-bit debuggee because the source address needs 64 bits.

PTT READ SPRS

This request writes the contents of the special purpose registers to the area specified by the Address parameter, which points to a ptsprs structure. The Identifier parameter specifies the traced kernel thread. The Data and Buffer parameters are ignored.

Note: For the PTT READ SPRS request, use ptracex or ptrace64 with the 64-bit debuggee because the new ptxsprs structure must be used.

PT_REATT

This request allows a new debugger, with the proper permissions, to trace a process that was already traced by another debugger. The *Identifier* parameter is interpreted as the process ID of the traced process. The Address, Data, and Buffer parameters are ignored.

If this request is unsuccessful, -1 is returned and the errno global variable is set to one the following codes:

ESRCH

The *Identifier* is not valid; or the traced process is a kernel process.

EPERM

Real or effective user ID of the debugger does not match that of the traced process, or the debugger does not have root authority.

EINVAL

The debugger and the traced process are the same.

PT REGSET

This request writes the contents of all 32 general purpose registers to the area specified by the Address parameter. This area must be at least 128 bytes for the 32-bit debuggee or 256 bytes for the 64-bit debuggee. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Data* and *Buffer* parameters are ignored.

If this request is unsuccessful, -1 is returned and the errno global variable is set to the following code:

EIO The Address parameter points to a location outside of the allocated address space of the process.

Note: For the PT_REGSET request, use ptracex or ptrace64 with the 64-bit debuggee because 64-bit registers requiring 256 bytes are returned.

PT SET

This request sets an internal flag or capability. The Data parameter indicates which flags are set. The following flag can be set:

PTFLAG FAST TRAP

Enables the special handling of a fast trap instruction ("Fast Trap Instructions" on page 980). When a fast trap instruction is executed in a process that has a signal handler for SIGTRAP, the signal handler will be called even if the process is being traced.

The *Identifier* parameter specifies the process ID of the traced process. The *Address* parameter, Buffer parameter, and the unused bits in the Data parameter are reserved for future use and should be set to 0.

PT TRACE ME

This request must be issued by the debugged process to be traced. Upon receipt of a signal, this request sets the process trace flag, placing the process in a stopped state, rather than the action specified by the **sigaction** subroutine. The *Identifier*, *Address*, *Data*, and Buffer parameters are ignored. Do not issue this request if the parent process does not expect to trace the debugged process.

As a security measure, the ptrace subroutine inhibits the set-user-ID facility on subsequent **exec** subroutines, as shown in the following example:

```
if((childpid = fork()) == 0)
{ /* child process */
ptrace(PT_TRACE_ME,0,0,0,0);
   execlp(
                  )/* your favorite exec*/
else
     /* parent
     /* wait for child to stop
       rc = wait(status)
```

Note: This is the only request that should be performed by the child. The parent should perform all other requests when the child is in a stopped state.

If this request is unsuccessful, -1 is returned and the errno global variable is set to the following code:

ESRCH

Process is debugged by a process that is not its parent.

PT WRITE BLOCK

This request writes a block of data into the debugged process address space. The Address parameter points to the location in the process address space to be written into. The Data parameter gives the length of the block in bytes, and must not be greater than 1024. The *Identifier* parameter is interpreted as the process ID of the traced process. The Buffer parameter points to the location in the debugging process address space where the data is copied. Upon successful completion, the value of the Data parameter is returned to the debugging process.

If this request is unsuccessful, -1 is returned and the errno global variable is set to one of the following codes:

EIO The Data parameter is less than 1 or greater than 1024.

EIO The Address parameter is not a valid pointer into the debugged process address space.

EFAULT

The Buffer parameter does not point to a readable location in the debugging process address space.

Note: For the PT WRITE BLOCK request, use ptracex or ptrace64 with the 64-bit debuggee because 64-bit registers requiring 256 bytes are returned.

PT_WRITE_FPR

This request sets the floating-point register specified by the Data parameter to the value specified by the Address parameter. The Identifier parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, -1 is returned and the errno global variable is set to the following code:

EIO The *Data* parameter is not a valid floating-point register. The *Data* parameter must be in the range 256-287.

PTT WRITE FPRS

This request updates the contents of the 32 floating point registers with the values specified in the area designated by the Address parameter. This area must be at least 256 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and Buffer parameters are ignored.

PT WRITE GPR

This request stores the value of the Data parameter in one of the process general-purpose or special-purpose registers. The Address parameter specifies the register to be modified. Upon successful completion, the value of the Data parameter is returned to the debugging process. The *Identifier* parameter is interpreted as the process ID of the traced process. The Buffer parameter is ignored.

Note: If ptracex or ptrace64 are used with a 64-bit debuggee for the PT_WRITE_GPR request, the new register value is NOT passed via the Data parameter, but is instead passed via the 8-byte area pointed to by the buffer parameter.

If this request is unsuccessful, -1 is returned and the errno global variable is set to the following code:

EIO The Address parameter is not a valid general-purpose or special-purpose register. The Address parameter must be in the range 0-31 or 128-136.

PTT WRITE GPRS

This request updates the contents of the 32 general purpose registers with the values specified in the area designated by the Address parameter. This area must be at least 128 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and Buffer parameters are ignored.

Note: For the **PTT_WRITE_GPRS** request, use **ptracex** or **ptrace64** with the 64-bit debuggee because 64-bit registers requiring 256 bytes are returned. The buffer points to long long source area.

PT_WRITE_I or PT_WRITE_D

These requests write the value of the *Data* parameter into the address space of the debugged process at the word-aligned address specified by the *Address* parameter. On all machines currently supported by AIX Version 4, instruction and data address spaces are not separated. The **PT_WRITE_I** and **PT_WRITE_D** instruction and data requests can be used with equal results. Upon successful completion, the value written into the address space of the debugged process is returned to the debugging process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, -1 is returned and the **errno** global variable is set to the following code:

The *Address* parameter points to a location in a pure procedure space and a copy cannot be made; the *Address* is not word-aligned; or, the *Address* is not valid. User blocks, kernel segments, and kernel extension segments are not considered valid addresses.

Note: For the or **PT_WRITE_I** or **PT_WRITE_D** request, use **ptracex** or **ptrace64** with a 64-bit debuggee because the target address needs 64 bits.

PTT WRITE SPRS

This request updates the special purpose registers with the values in the area specified by the *Address* parameter, which points to a **ptsprs** structure. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

Identifier

Determined by the value of the Request parameter.

Address

Determined by the value of the *Request* parameter.

Data Determined by the value of the Request parameter.

Buffer Determined by the value of the Request parameter.

Note: For the **PTT_READ_SPRS** request, use **ptracex** or **ptrace64** with the 64-bit debuggee because the new **ptxsprs** structure must be used.

Error Codes

The **ptrace** subroutine is unsuccessful when one of the following is true:

EFAULT The *Buffer* parameter points to a location outside the debugging process address space.

EINVAL The debugger and the traced process are the same; or the *Identifier* parameter does not identify the

thread that caused the exception.

The *Request* parameter is not one of the values listed, or the *Request* parameter is not valid for the

machine type on which the process is executed.

ENOMEM Either the area is not large enough to accommodate the loader information, or there is not enough

memory to allocate an equivalent buffer in the kernel.

EPERM The *Identifier* parameter corresponds to a kernel thread which is stopped in kernel mode and whose

computational state cannot be read or written.

ESRCH The Identifier parameter identifies a process or thread that does not exist, that has not executed a ptrace

call with the $\mbox{\bf PT_TRACE_ME}$ request, or that is not stopped.

For ptrace: If the debuggee is a 64-bit process, the options that refer to GPRs or SPRs fail with errno = **EIO**, and the options that specify addresses are limited to 32-bits.

For ptracex or ptrace64: If the debuggee is a 32-bit process, the options that refer to GPRs or SPRs fail with errno = EIO, and the options that specify addresses in the debuggee's address space that are larger than 2**32 - 1 fail with errno set to EIO.

Also, the options PT_READ_U and PT_WRITE_U are not supported if the debuggee is a 64-bit program (errno = **ENOTSUP**).

Related Information

The "exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196, "getprocs Subroutine" on page 351, "getthrds Subroutine" on page 372, and "load Subroutine" on page 618.

The sigaction subroutine, unload subroutine, and wait, waitpid, or wait3 subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

The dbx command AIX 5L Version 5.2 Commands Reference, Volume 2.

ptsname Subroutine

Purpose

Returns the name of a pseudo-terminal device.

Library

Standard C Library (libc.a)

Syntax

#include <stdlib.h>

char *ptsname (FileDescriptor) int FileDescriptor

Description

The ptsname subroutine gets the path name of the slave pseudo-terminal associated with the master pseudo-terminal device defined by the FileDescriptor parameter.

Parameters

FileDescriptor

Specifies the file descriptor of the master pseudo-terminal device

Return Values

The **ptsname** subroutine returns a pointer to a string containing the null-terminated path name of the pseudo-terminal device associated with the file descriptor specified by the FileDescriptor parameter. A null pointer is returned and the errno global variable is set to indicate the error if the file descriptor does not describe a pseudo-terminal device in the /dev directory.

Files

/dev/*

Terminal device special files.

Related Information

The **ttyname** subroutine.

The Input and Output Handling Programmer's Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

putc, putchar, fputc, or putw Subroutine

Purpose

Writes a character or a word to a stream.

Library

Standard I/O Package (libc.a)

Syntax

```
#include <stdio.h>
int putc ( Character, Stream)
int Character;
FILE *Stream;
int putchar (Character)
int Character;
int fputc (Character, Stream)
int Character:
FILE *Stream;
int putw ( Word, Stream)
int Word;
FILE *Stream;
```

Description

The putc and putchar macros write a character or word to a stream. The fputc and putw subroutines serve similar purposes but are true subroutines.

The putc macro writes the character Character (converted to an unsigned char data type) to the output specified by the Stream parameter. The character is written at the position at which the file pointer is currently pointing, if defined.

The putchar macro is the same as the putc macro except that putchar writes to the standard output.

The fputc subroutine works the same as the putc macro, but fputc is a true subroutine rather than a macro. It runs more slowly than putc, but takes less space per invocation.

Because **putc** is implemented as a macro, it incorrectly treats a *Stream* parameter with side effects, such as putc(C, *f++). For such cases, use the fputc subroutine instead. Also, use fputc whenever you need to pass a pointer to this subroutine as a parameter to another subroutine.

The putc and putchar macros have also been implemented as subroutines for ANSI compatibility. To access the subroutines instead of the macros, insert #undef putc or #undef putchar at the beginning of the source file.

The putw subroutine writes the word (int data type) specified by the Word parameter to the output specified by the Stream parameter. The word is written at the position at which the file pointer, if defined, is pointing. The size of a word is the size of an integer and varies from machine to machine. The putw subroutine does not assume or cause special alignment of the data in the file.

After the fputcw, putce, fputc, putc, fputs, puts, or putw subroutine runs successfully, and before the next successful completion of a call either to the fflush or fclose subroutine on the same stream or to the exit or abort subroutine, the st ctime and st mtime fields of the file are marked for update.

Because of possible differences in word length and byte ordering, files written using the putw subroutine are machine-dependent, and may not be readable using the getw subroutine on a different processor.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line-buffered if they refer to terminals. The standard error output stream, stderr, is unbuffered by default, but using the freopen subroutine causes it to become buffered or line-buffered. Use the setbuf subroutine to change the stream buffering strategy.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as it is written. When an output stream is buffered, many characters are saved and written as a block. When an output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested).

Parameters

Stream Points to the file structure of an open file. Character Specifies a character to be written.

Word Specifies a word to be written (not portable because word length and byte-ordering are

machine-dependent).

Return Values

Upon successful completion, these functions each return the value written. If these functions fail, they return the constant EOF. They fail if the Stream parameter is not open for writing, or if the output file size cannot be increased. Because the EOF value is a valid integer, you should use the ferror subroutine to detect **putw** errors.

Error Codes

The **fputc** subroutine will fail if either the *Stream* is unbuffered or the *Stream* buffer needs to be flushed, and:

EAGAIN The O_NONBLOCK flag is set for the file descriptor underlying Stream and the process would be delayed in the write operation.

EBADF The file descriptor underlying Stream is not a valid file descriptor open for writing.

EFBIG An attempt was made to write a file that exceeds the file size of the process limit or the maximum file

size.

EFBIG The file is a regular file and an attempt was made to write at or beyond the offset maximum.

The write operation was terminated due to the receipt of a signal, and either no data was transferred or **EINTR**

the implementation does not report partial transfers for this file.

Note: Depending upon which library routine the application binds to, this subroutine may return EINTR.

Refer to the signal Subroutine regarding sa_restart.

EIO A physical I/O error has occurred, or the process is a member of a background process group attempting

> to perform a write subroutine to its controlling terminal, the TOSTOP flag is set, the process is neither ignoring nor blocking the SIGTTOU signal and the process group of the process is orphaned. This error

may also be returned under implementation-dependent conditions.

ENOSPC There was no free space remaining on the device containing the file.

EPIPE An attempt is made to write to a pipe or first-in-first-out (FIFO) that is not open for reading by any

process. A SIGPIPE signal will also be sent to the process.

The **fputc** subroutine may fail if:

ENOMEM Insufficient storage space is available.

ENXIO A request was made of a nonexistent device, or the request was outside the capabilities of the device.

Related Information

The fclose or fflush ("fclose or fflush Subroutine" on page 213) subroutine, feof, ferror, clearerr, or fileno ("feof, ferror, clearerr, or fileno Macro" on page 227) subroutine, fopen, freopen, or fdopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, fread or fwrite ("fread or fwrite Subroutine" on page 267) subroutine, getc, fgetc, getchar, or getw ("getc, getchar, fgetc, or getw Subroutine" on page 300) subroutine, getwc, fgetwc, or getwchar ("getwc, fgetwc, or getwchar Subroutine" on page 397) subroutine, printf, fprintf, sprintf, NLprintf, NLsprintf, or wsprintf ("printf, fprintf, sprintf, sprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine, putwc, fputwc, or putwchar ("putwc, putwchar, or fputwc Subroutine" on page 995) subroutine, puts or fputs ("puts or fputs Subroutine" on page 994) subroutine, setbuf subroutine.

putenv Subroutine

Purpose

Sets an environment variable.

Library

Standard C Library (libc.a)

Syntax

int putenv (String) char *String;

Description

Attention: Unpredictable results can occur if a subroutine passes the putenv subroutine a pointer to an automatic variable and then returns while the variable is still part of the environment.

The **putery** subroutine sets the value of an environment variable by altering an existing variable or by creating a new one. The String parameter points to a string of the form Name=Value, where Name is the environment variable and Value is the new value for it.

The memory space pointed to by the *String* parameter becomes part of the environment, so that altering the string effectively changes part of the environment. The space is no longer used after the value of the environment variable is changed by calling the puterv subroutine again. Also, after the puterv subroutine is called, environment variables are not necessarily in alphabetical order.

The putenv subroutine manipulates the environ external variable and can be used in conjunction with the getenv subroutine. However, the EnvironmentPointer parameter, the third parameter to the main subroutine, is not changed.

The **puterv** subroutine uses the **malloc** subroutine to enlarge the environment.

Parameters

String A pointer to the Name=Value string.

Return Values

Upon successful completion, a value of 0 is returned. If the malloc subroutine is unable to obtain sufficient space to expand the environment, then the putenv subroutine returns a nonzero value.

Related Information

The exec: execl, execv, execle, execlp, execvp, or exect ("exec: execl, execle, execlp, execv, execve, execvp, or exect Subroutine" on page 196) subroutine, getenv ("getenv Subroutine" on page 313) subroutine, malloc ("malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo heap, alloca, or valloc Subroutine" on page 657) subroutine.

putgrent Subroutine

Purpose

Updates group descriptions.

Library

Standard C Library (libc.a)

Syntax

int putgrent (grp, fp) struct group *grp; FILE *fp;

Description

The **putgrent** subroutine updates group descriptions. The *grp* parameter is a pointer to a group structure, as created by the getgrent, getgrgid, and getgrnam subroutines.

The **putgrent** subroutine writes a line on the stream specified by the fp parameter. The stream matches the format of /etc/group.

The **gr_passwd** field of the line written is always set to ! (exclamation point).

Parameters

Pointer to a group structure. grp

Specifies the stream to be written to. fp

Return Values

The putgrent subroutine returns a value of 0 upon successful completion. If putgrent fails, a nonzero value is returned.

Files

/etc/group

/etc/security/group

Related Information

The "getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine" on page 318.

List of Security and Auditing Subroutines, Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

puts or fputs Subroutine

Purpose

Writes a string to a stream.

Library

Standard I/O Library (libc.a)

Syntax

```
#include <stdio.h>
int puts (String)
const char *String;
int fputs (String, Stream)
const char *String;
FILE *Stream;
```

Description

The **puts** subroutine writes the string pointed to by the *String* parameter to the standard output stream, **stdout**, and appends a new-line character to the output.

The **fputs** subroutine writes the null-terminated string pointed to by the *String* parameter to the output stream specified by the *Stream* parameter. The **fputs** subroutine does not append a new-line character.

Neither subroutine writes the terminating null character.

After the **fputwc**, **putwc**, **fputc**, **fputs**, **puts**, or **putw** subroutine runs successfully, and before the next successful completion of a call either to the **fflush** or **fclose** subroutine on the same stream or a call to the **exit** or **abort** subroutine, the st ctime and st mtime fields of the file are marked for update.

Parameters

String Points to a string to be written to output.

Stream Points to the FILE structure of an open file.

Return Values

Upon successful completion, the **puts** and **fputs** subroutines return the number of characters written. Otherwise, both subroutines return **EOF**, set an error indicator for the stream and set the **errno** global variable to indicate the error. This happens if the routines try to write to a file that has not been opened for writing.

Error Codes

If the **puts** or **fputs** subroutine is unsuccessful because the output stream specified by the *Stream* parameter is unbuffered or the buffer needs to be flushed, it returns one or more of the following error codes:

EAGAIN Indicates that the **O_NONBLOCK** flag is set for the file descriptor specified by the *Stream* parameter and

the process would be delayed in the write operation.

EBADF Indicates that the file descriptor specified by the *Stream* parameter is not a valid file descriptor open for

writing.

EFBIG Indicates that an attempt was made to write to a file that exceeds the process' file size limit or the

systemwide maximum file size.

EINTR Indicates that the write operation was terminated due to receipt of a signal and no data was transferred.

Note: Depending upon which library routine the application binds to, this subroutine may return EINTR.

Refer to the signal subroutine regarding the SA_RESTART bit.

EIO Indicates that the process is a member of a background process group attempting to perform a write to

its controlling terminal, the TOSTOP flag is set, the process is neither ignoring or blocking the SIGTTOU

signal, and the process group of the process has no parent process.

ENOSPC Indicates that there was no free space remaining on the device containing the file specified by the

Stream parameter.

EPIPE Indicates that an attempt is made to write to a pipe or first-in-first-out (FIFO) that is not open for reading

by any process. A SIGPIPE signal will also be sent to the process.

ENOMEM Indicates that insufficient storage space is available.

ENXIO Indicates that a request was made of a nonexistent device, or the request was outside the capabilities of

the device.

Related Information

The **fopen**, **freopen**, or **fdopen** ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, **fread**, or **fwrite** ("fread or fwrite Subroutine" on page 267) subroutine, **gets** or **fgets** ("gets or fgets Subroutine" on page 366) subroutine, **getws** or **fgetws** ("getws or fgetws Subroutine" on page 399) subroutine, **printf**, **fprintf**, and **sprintf** ("printf, fprintf, sprintf, sprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine, **putc**, **putchar**, **fputc**, or **putw** ("putc, putchar, fputc, or putw Subroutine" on page 990)subroutine, **putws**, **putwchar**, or **fputwc** ("putwc, putwchar, or fputwc Subroutine") subroutine, **putws** or **fputws** ("putws or fputws Subroutine" on page 997) subroutine.

The **feof**, **ferror**, **clearerr**, or **fileno** ("feof, ferror, clearerr, or fileno Macro" on page 227) macros.

List of String Manipulation Services.

Subroutines Overview in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

putwc, putwchar, or fputwc Subroutine

Purpose

Writes a character or a word to a stream.

Library

Standard I/O Library (libc.a)

Syntax

```
#include <stdio.h>
wint t putwc( Character, Stream)
wint t Character;
FILE *Stream;
wint t putwchar(Character)
wint_t Character;
wint t fputwc(Character, Stream)
wint t Character;
FILE Stream;
```

Description

The **putwc** subroutine writes the wide character specified by the *Character* parameter to the output stream pointed to by the Stream parameter. The wide character is written as a multibyte character at the associated file position indicator for the stream, if defined. The subroutine then advances the indicator. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

The putwchar subroutine works like the putwc subroutine, except that putwchar writes the specified wide character to the standard output.

The **fputwc** subroutine works the same as the **putwc** subroutine.

Output streams, with the exception of stderr, are buffered by default if they refer to files, or line-buffered if they refer to terminals. The standard error output stream, stderr, is unbuffered by default, but using the freopen subroutine causes it to become buffered or line-buffered. Use the setbuf subroutine to change the stream's buffering strategy.

After the fputwc, putwc, fputc. putc, fputs, puts, or putw subroutine runs successfully, and before the next successful completion of a call either to the fflush or fclose subroutine on the same stream or to the exit or abort subroutine, the st ctime and st mtime fields of the file are marked for update.

Parameters

Character Specifies a wide character of type wint_t. Stream Specifies a stream of output data.

Return Values

Upon successful completion, the putwc, putwchar, and fputwc subroutines return the wide character that is written. Otherwise WEOF is returned, the error indicator for the stream is set, and the errno global variable is set to indicate the error.

Error Codes

If the putwc, putwchar, or fputwc subroutine fails because the stream is not buffered or data in the buffer needs to be written, it returns one or more of the following error codes:

EAGAIN Indicates that the O_NONBLOCK flag is set for the file descriptor underlying the Stream parameter, delaying the process during the write operation.

EBADF Indicates that the file descriptor underlying the Stream parameter is not valid and cannot be updated

during the write operation.

Indicates that the process attempted to write to a file that already equals or exceeds the file-size limit for **EFBIG**

the process. The file is a regular file and an attempt was made to write at or beyond the offset maximum

associated with the corresponding stream.

EILSEQ Indicates that the wide-character code does not correspond to a valid character. **EINTR** Indicates that the process has received a signal that terminates the read operation.

Indicates that the process is in a background process group attempting to perform a write operation to its **EIO**

controlling terminal. The TOSTOP flag is set, the process is not ignoring or blocking the SIGTTOU flag,

and the process group of the process is orphaned.

Insufficient storage space is available. **ENOMEM**

ENOSPC Indicates that no free space remains on the device containing the file.

ENXIO Indicates a request was made of a non-existent device, or the request was outside the capabilities of the

device.

EPIPE Indicates that the process has attempted to write to a pipe or first-in-first-out (FIFO) that is not open for

reading. The process will also receive a SIGPIPE signal.

Related Information

Other wide character I/O subroutines: fgetwc ("getwc, fgetwc, or getwchar Subroutine" on page 397) subroutine, fgetws ("getws or fgetws Subroutine" on page 399) subroutine, fputws ("putws or fputws Subroutine") subroutine, **getwc** ("getwc, fgetwc, or getwchar Subroutine" on page 397) subroutine, getwchar ("getwc, fgetwc, or getwchar Subroutine" on page 397) subroutine, getws ("getws or fgetws Subroutine" on page 399) subroutine, putws ("putws or fputws Subroutine") subroutine, ungetwc subroutine.

Related standard I/O subroutines: fdopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, fgets ("gets or fgets Subroutine" on page 366) subroutine, fopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, fprintf ("printf, fprintf, sprintf, sprintf, wsprintf, vprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine, fputc ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine, fputs ("puts or fputs Subroutine" on page 994) subroutine, fread ("fread or fwrite Subroutine" on page 267) subroutine, freopen ("fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245) subroutine, fwrite ("fread or fwrite Subroutine" on page 267) subroutine, gets ("gets or fgets Subroutine" on page 366) subroutine, printf ("printf, fprintf, sprintf, snprintf, wsprintf, vprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine, putc ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine, putchar ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine, puts ("puts or fputs Subroutine" on page 994) subroutine, putw ("putc, putchar, fputc, or putw Subroutine" on page 990) subroutine, sprintf ("printf, fprintf, sprintf, sprintf, wsprintf, vprintf, vsprintf, vsprintf, or vwsprintf Subroutine" on page 869) subroutine.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overviewand Multibyte Code and Wide Character Code Conversion Subroutines in AIX 5L Version 5.2 National Language Support Guide and Reference.

putws or fputws Subroutine

Purpose

Writes a wide-character string to a stream.

Library

Standard I/O Library (libc.a)

Syntax

```
#include <stdio.h>
int putws ( String)
const wchar_t *String;
int fputws (String, Stream)
const wchar_t *String;
FILE *Stream;
```

Description

The **putws** subroutine writes the **const wchar_t** string pointed to by the *String* parameter to the standard output stream (**stdout**) as a multibyte character string and appends a new-line character to the output. In all other respects, the **putws** subroutine functions like the **puts** subroutine.

The **fputws** subroutine writes the **const wchar_t** string pointed to by the *String* parameter to the output stream as a multibyte character string. In all other respects, the **fputws** subroutine functions like the **fputs** subroutine.

After the **putws** or **fputws** subroutine runs successfully, and before the next successful completion of a call to the **fflush** or **fclose** subroutine on the same stream or a call to the **exit** or **abort** subroutine, the st ctime and st mtime fields of the file are marked for update.

Parameters

String Points to a string to be written to output.

Stream Points to the FILE structure of an open file.

Return Values

Upon successful completion, the **putws** and **fputws** subroutines return a nonnegative number. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **putws** or **fputws** subroutine is unsuccessful if the stream is not buffered or data in the buffer needs to be written, and one of the following errors occur:

EAGAIN	The O_NONBLOCK flag is set for the file descriptor underlying the Stream parameter, which delays the
---------------	--

process during the write operation.

EBADF The file descriptor underlying the *Stream* parameter is not valid and cannot be updated during the write

operation.

EFBIG The process attempted to write to a file that already equals or exceeds the file-size limit for the process.

EINTR The process has received a signal that terminates the read operation.

EIO The process is in a background process group attempting to perform a write operation to its controlling

terminal. The **TOSTOP** flag is set, the process is not ignoring or blocking the **SIGTTOU** flag, and the

process group of the process is orphaned.

ENOSPC No free space remains on the device containing the file.

EPIPE The process has attempted to write to a pipe or first-in-first-out (FIFO) that is not open for reading. The

process also receives a SIGPIPE signal.

EILSEQ The **wc** wide-character code does not correspond to a valid character.

Related Information

Other wide-character I/O subroutines: "getwc, fgetwc, or getwchar Subroutine" on page 397, "getws or fgetws Subroutine" on page 399, "putwc, putwchar, or fputwc Subroutine" on page 995, and ungetwc subroutine.

Related standard I/O subroutines: "fopen, fopen64, freopen, freopen64 or fdopen Subroutine" on page 245, "gets or fgets Subroutine" on page 366, "printf, fprintf, sprintf, snprintf, wsprintf, vprintf, vfprintf, vsprintf, or vwsprintf Subroutine" on page 869, "putc, putchar, fputc, or putw Subroutine" on page 990, "puts or fputs Subroutine" on page 994, "fread or fwrite Subroutine" on page 267.

Subroutines, Example Programs, and Libraries in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

National Language Support Overview and Multibyte Code and Wide Character Code Conversion Subroutines in AIX 5L Version 5.2 National Language Support Guide and Reference.

pwdrestrict method Subroutine

Purpose

Defines loadable password restriction methods.

Library

Syntax

```
int pwdrestrict method (UserName, NewPassword, OldPassword, Message)
char * UserName:
char * NewPassword;
char * OldPassword;
char ** Message;
```

Description

The pwdrestrict_method subroutine extends the capability of the password restrictions software and lets an administrator enforce password restrictions that are not provided by the system software.

Whenever users change their passwords, the system software scans the pwdchecks attribute defined for that user for site specific restrictions. Since this attribute field can contain load module file names, for example, methods, it is possible for the administrator to write and install code that enforces site specific password restrictions.

The system evaluates the pwdchecks attribute's value field in a left to right order. For each method that the system encounters, the system loads and invokes that method. The system uses the load subroutine to load methods. It invokes the **load** subroutine with a Flags value of 1 and a LibraryPath value of /usr/lib. Once the method is loaded, the system invokes the method.

To create a loadable module, use the -e flag of the Id command. Note that the name pwdrestrict_method given in the syntax is a generic name. The actual subroutine name can be anything (within the compiler's name space) except main. What is important is, that for whatever name you choose, you must inform the Id command of the name so that the load subroutine uses that name as the entry point into the module. In the following example, the C compiler compiles the pwdrestrict.c file and pass -e pwdrestrict method to the **Id** command to create the method called **pwdrestrict**:

```
cc -e pwdrestrict method -o pwdrestrict pwdrestrict.c
```

The convention of all password restriction methods is to pass back messages to the invoking subroutine. Do not print messages to stdout or stderr. This feature allows the password restrictions software to work across network connections where stdout and stderr are not valid. Note that messages must be returned in dynamically allocated memory to the invoking program. The invoking program will deallocate the memory once it is done with the memory.

There are many caveats that go along with loadable subroutine modules:

- 1. The values for NewPassword and OldPassword are the actual clear text passwords typed in by the user. If you copy these passwords into other parts of memory, clear those memory locations before returning back to the invoking program. This helps to prevent clear text passwords from showing up in core dumps. Also, do not copy these passwords into a file or anywhere else that another program can access. Clear text passwords should never exist outside of the process space.
- 2. Do not modify the current settings of the process' signal handlers.
- 3. Do not call any functions that will terminate the execution of the program (for example, the exit subroutine, the **exec** subroutine). Always return to the invoking program.
- 4. The code must be thread-safe.
- 5. The actual load module must be kept in a write protected environment. The load module and directory should be writable only by the root user.

One last note, all standard password restrictions are performed before any of the site specific methods are invoked. Thus, methods are the last restrictions to be enforced by the system.

Parameters

UserName Specifies a "local" user name.

NewPassword Specifies the new password in clear text (not encrypted). This value may be a NULL pointer.

Clear text passwords are always in 7 bit ASCII.

OldPassword Specifies the current password in clear text (not encrypted). This value may be a NULL pointer.

Clear text passwords are always in 7 bit ASCII.

Specifies the address of a pointer to malloc'ed memory containing an NLS error message. The Message

method is expected to supply the **malloc**'ed memory and the message.

Return Values

The method is expected to return the following values. The return values are listed in order of precedence.

- -1 Internal error. The method could not perform its password evaluation. The method must set the **errno** variable. The method must supply an error message in Message unless it can't allocate memory for the message. If it cannot allocate memory, then it must return the NULL pointer in Message.
- 1 Failure. The password change did not meet the requirements of the restriction. The password restriction was properly evaluated and the password change was not accepted. The method must supply an error message in Message. The errno variable is ignored. Note that composition failures are cumulative, thus, even though a failure condition is returned, trailing composition methods will be invoked.
- 0 Success. The password change met the requirements of the restriction. If necessary, the method may supply a message in *Message*; otherwise, return the NULL pointer. The **errno** variable is ignored.

Appendix A. Base Operating System Error Codes for Services That Require Path-Name Resolution

The following errors apply to any service that requires path name resolution:

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

EFAULT The *Path* parameter points outside of the allocated address space of the process.

EIO An I/O error occurred during the operation.

ELOOP Too many symbolic links were encountered in translating the *Path* parameter. **ENAMETOOLONG** A component of a path name exceeded 255 characters and the process has the

DisallowTruncation attribute (see the ulimit subroutine) or an entire path name exceeded

1023 characters.

ENOENT A component of the path prefix does not exist.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ENOENT The path name is null.

ENOTDIR A component of the path prefix is not a directory.

ESTALE The root or current directory of the process is located in a virtual file system that is

unmounted.

Related Information

List of File and Directory Manipulation Services.

© Copyright IBM Corp. 1994, 2004

Appendix B. ODM Error Codes

When an ODM subroutine is unsuccessful, a value of -1 is returned and the odmerrno variable is set to one of the following values:

ODMI_BAD_CLASSNAME The specified object class name does not match the object class name in the

file. Check path name and permissions.

ODMI_BAD_CLXNNAME The specified collection name does not match the collection name in the file. ODMI_BAD_CRIT

The specified search criteria is incorrectly formed. Make sure the criteria contains only valid descriptor names and the search values are correct. For information on qualifying criteria, see "Understanding ODM Object Searches"

in AIX 5L Version 5.2 General Programming Concepts: Writing and

Debugging Programs.

ODMI BAD LOCK Cannot set a lock on the file. Check path name and permissions. ODMI BAD TIMEOUT The time-out value was not valid. It must be a positive integer.

ODMI_BAD_TOKEN Cannot create or open the lock file. Check path name and permissions. ODMI CLASS DNE The specified object class does not exist. Check path name and permissions. The specified object class already exists. An object class must not exist when ODMI CLASS EXISTS

it is created.

ODMI_CLASS_PERMS The object class cannot be opened because of the file permissions.

ODMI_CLXNMAGICNO_ERR The specified collection is not a valid object class collection. **ODMI FORK**

Cannot fork the child process. Make sure the child process is executable and

try again.

ODMI_INTERNAL_ERR An internal consistency problem occurred. Make sure the object class is valid

or contact the person responsible for the system.

The specified file is not an object class. ODMI_INVALID_CLASS

ODMI_INVALID_CLXN Either the specified collection is not a valid object class collection or the

collection does not contain consistent data.

ODMI_INVALID_PATH The specified path does not exist on the file system. Make sure the path is

accessible.

ODMI_LINK_NOT_FOUND The object class that is accessed could not be opened. Make sure the linked

object class is accessible.

ODMI LOCK BLOCKED Cannot grant the lock. Another process already has the lock.

Cannot retrieve or set the lock environment variable. Remove some ODMI_LOCK_ENV

environment variables and try again.

The lock identifier does not refer to a valid lock. The lock identifier must be ODMI_LOCK_ID

the same as what was returned from the odm_lock ("odm_lock Subroutine"

on page 752) subroutine.

ODMI MAGICNO ERR The class symbol does not identify a valid object class.

Cannot allocate sufficient storage. Try again later or contact the person ODMI_MALLOC_ERR

responsible for the system.

ODMI NO OBJECT The specified object identifier did not refer to a valid object.

Cannot open the object class. Check path name and permissions. ODMI_OPEN_ERR ODMI_OPEN_PIPE Cannot open a pipe to a child process. Make sure the child process is

executable and try again.

ODMI PARAMS The parameters passed to the subroutine were not correct. Make sure there

are the correct number of parameters and that they are valid.

ODMI READ ONLY The specified object class is opened as read-only and cannot be modified.

Cannot read from the pipe of the child process. Make sure the child process

is executable and try again.

ODMI_READ_PIPE

ODMI TOOMANYCLASSES Too many object classes have been accessed. An application can only

access less than 1024 object classes.

ODMI_UNLINKCLASS_ERR Cannot remove the object class from the file system. Check path name and

permissions.

Cannot remove the object class collection from the file system. Check path ODMI_UNLINKCLXN_ERR

name and permissions.

1003 © Copyright IBM Corp. 1994, 2004

Related Information

List of ODM Commands and Subroutines in AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs.

Appendix C. 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 give 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.

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.

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:

IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
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 document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

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 Licensina 2-31 Roppongi 3-chome, Minato-ku Tokyo 106, Japan

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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.

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.

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 illustrates 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. 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 IBM's application programming interfaces.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters	access control subroutines (continued)
_atojis macro 474	acl_fput 13
_check_lock Subroutine 104	acl_fset 15
_clear_lock Subroutine 104	acl_get 11
edata identifier 184	acl_put 13
_end identifier 184	acl_set 15
_exit subroutine 203	chacl 119
Exit subroutine 203	chmod 124
extext identifier 184	chown 127
iistoa macro 474	chownx 127
 _lazySetErrorHandler Subroutine 482	fchacl 119
_tojlower macro 474	fchmod 124
_tojupper macro 474	fchown 127 fchownx 127
_tolower subroutine 149	frevoke 270
_toupper subroutine 149	access subroutine 5
/etc/filesystems file	accessx subroutine 5
accessing entries 316	acct subroutine 8
/etc/hosts file	acl_chg subroutine 9
closing 740	acl_fchg subroutine 9
retrieving host entries 739	acl_fget subroutine 11
/etc/utmp file	acl_fput subroutine 13
accessing entries 393	acl_fset subroutine 15
	acl_get subroutine 11
Numerics	acl_put subroutine 13
	acl_set subroutine 15
164a_r subroutine 2	acos subroutine 17
199332 383	acosf subroutine 17
3-byte integers	acosh subroutine 18
converting 484	acoshf subroutine 18
	acoshl subroutine 18
Α	acosl subroutine 17
	address identifiers 184
a64l subroutine 1 abort subroutine 3	addssys subroutine 19
abs subroutine 4	adjtime subroutine 21 advance subroutine 144
absinterval subroutine 329	aio cancel subroutine 22
absolute path names	aio_error subroutine 25
copying 398	aio_fsync subroutine 28
determining 398	aio_nwait subroutine 29
absolute value subroutines	aio_nwait_timeout subroutine 31
cabs 105	aio_read subroutine 33
cabsf 105	aio_return subroutine 37
cabsl 105	aio_suspend subroutine 40
fabsf 209	aio_write subroutine 43
absolute values	alarm subroutine 329
computing complex 426	alloca subroutine 657
imaxabs 433	Application Programming Interface
access control attributes	perfstat
setting 119	cpu 798
access control information	cpu_total 799, 806, 810
changing 9	disk_total 801, 805, 808
retrieving 11	diskpath 803
setting 13, 15	netbuffer 807
access control subroutines	pagingspace 811
acl_chg 9	protocol 812
acl_fchg 9 acl_fget 11	reset 814
au_iyet ii	

arc sine subroutines	atomic access subroutines (continued)
asinf 72	fetch_and_add 229
arc tangent subroutines	fetch_and_and 230
atan2f 73	fetch_and_or 230
atan2l 73	audit bin files
atanf 75	compressing and uncompressing 89
atani 75	establishing 81
archive files	audit records
reading headers 589	generating 85
ARM Subroutines	reading 92
arm_end 50	writing 93
arm_end Dual Call 51	audit subroutine 79
arm_getid 53	audit trail files
arm_getid Dual Call 55	appending records 85
arm_init 58	auditbin subroutine 81
arm_init Dual Call 60	auditevents subroutine 83
arm_start 61	auditing modes 86
arm_start Dual Call 63	auditing subroutines
arm_stop 65	audit 79
arm_stop Dual Call 66	auditbin 81
arm_update 68	auditevents 83
arm_update Dual Call 69	auditlog 85
ASCII strings	auditobj 86
converting to floating-point numbers 77	auditpack 89
converting to Internet addresses 455	auditproc 90
asctime subroutine 163	auditread 92
asctime_r subroutine 166	auditwrite 93
asin subroutine 72	auditlog subroutine 85
asinf subroutine 72	auditobj subroutine 86
asinh subroutine 71	auditpack subroutine 89
asinhf subroutine 71	auditproc subroutine 90
asinhl subroutine 71	auditread, auditread_r subroutines 92
asinl subroutine 72	auditwrite subroutine 93
assert macro 73	authenticate 94
asynchronous I/O	authentication subroutines
reading 33	ckuseracct 135
writing 43	ckuserID 137
asynchronous I/O requests	crypt 157
canceling 22	encrypt 157
listing 611	getlogin 335
retrieving error status 25	getpass 340
retrieving return status 37	getuserpw 389
suspending 40	newpass 732
synchronizing asynchronous files 28	putuserpw 389
atan subroutine 75	setkey 157
atan2 subroutine 73	authorizations 389
atan2f subroutine 73	authorizations, compare 667
atan2l subroutine 73	auxiliary areas
atanf subroutine 75	creating 435
atanh subroutine 75	destroying 435
atanhf subroutine 75	drawing 436
atanhl subroutine 75	hiding 437
atanl subroutine 75	processing 449
atexit subroutine 203	
atof subroutine 77	D
atoff subroutine 77	В
atojis subroutine 474	base 10 logarithm functions
atol subroutine 78	log10f 631
atoll subroutine 78	base 2 logarithm functions
atomic access subroutines	log2 633
compare_and_swap 143	log2f 633

base 2 logarithm functions (continued)	ccosf subroutine 114
log2l 633	ccosh subroutine 115
basename Subroutine 96	ccoshf subroutine 115
baud rates	ccoshl subroutine 115
getting and setting 118	ccosl subroutine 114
bcmp subroutine 97	CCSIDs
bcopy subroutine 97	converting 115
beep levels	ccsidtocs subroutine 115
setting 437	ceil subroutine 116
BeginCriticalSection Subroutine 186	ceilf subroutine 116
Bessel functions	ceiling value function
computing 98	ceilf 116
binary files	ceill 116
reading 267	ceill subroutine 116
-	cexp subroutine 117
,	
binding a process to a processor 99	cexpf subroutine 117
bit string operations 97	cexpl subroutine 117
box characters	cfgetospeed subroutine 118
shaping 584	chacl subroutine 119
brk subroutine 100	character conversion
bsearch subroutine 102	8-bit processing codes and 473
btowc subroutine 103	code set converters 429, 430
buffered data	conv subroutines 149
writing to streams 213	Japanese 474
byte string operations 97	Kanji-specific 473
bzero subroutine 97	multibyte to wide 680, 681
	translation operations 149
	character manipulation subroutines
C	_atojis 474
	_iistoa 474
cabs subroutine 105	
cabsf subroutine 105	_tojlower 474
cabsl subroutine 105	_tojupper 474
cacos subroutine 105	_tolower 149
cacosf subroutine 105	_toupper 149
cacosh subroutines 106	atojis 474
cacoshf subroutine 106	conv 149
cacoshl subroutine 106	ctype 476
cacosl subroutine 105	fgetc 300
calloc subroutine 657	fputc 990
carg subroutine 107	getc 300
cargf subroutine 107	getchar 300
cargl subroutine 107	getw 300
casin subroutine 107	isalnum 168
casinf subroutine 107	isalpha 168
casinf subroutine 108	isascii 168
	iscntrl 168
casinh subroutines 108	isdigit 168
casinl subroutine 107	isgraph 168
casinlh subroutine 108	- ·
catan subroutine 109	isjalnum 476
catanf subroutine 109	isjalpha 476
catanh subroutine 109	isjdigit 476
catanhf subroutine 109	isjgraph 476
catanhl subroutine 109	isjhira 476
catanl subroutine 109	isjis 476
catclose subroutine 110	isjkanji 476
catgets subroutine 111	isjkata 476
catopen subroutine 112	isjlbytekana 476
cbrt subroutine 113	isjlower 476
cbrtf subroutine 113	isjparen 476
cbrtl subroutine 113	isjprint 476
	isjpunct 476
ccos, subroutine 114	iojpariot 170

character manipulation subroutines (continued)	code sets
isjspace 476	closing converters 429
isjupper 476	converting names 115
isjxdigit 476	opening converters 430
islower 168	coded character set IDs
isparent 476	converting 115
isprint 168	command-line flags
ispunct 168	returning 337
isspace 168	Common Host Bus Adapter library
isupper 168	HBA_SetRNIDMgmtInfo 423
isxdigit 168	compare_and_swap subroutine
jistoa 474	atomic access 143
kutentojis 474	compile subroutine 144
NCesc 149	complementary error subroutines
NCflatchr 149	erfcl 188
NCtolower 149	complex arc cosine subroutines
NCtoNLchar 149	cacos 105
NCtoupper 149	cacosf 105
NCunesc 149	cacosl 105
putc 990	complex arc hyperbolic cosine subroutines
putchar 990	cacosh 106
putw 990	cacoshf 106
toascii 149	cacoshl 106
tojhira 474	complex arc hyperbolic sine subroutines
tojkata 474	casin 108
tojlower 474	casinf 108
tojupper 474	casinI 108
tolower 149	complex arc hyperbolic tangent subroutines
toujis 474	catanh 109
toupper 149	catanhf 109
character shaping 577	catanhl 109
character testing	complex arc sine subroutines
isblank 462	casin 107
characters	casinf 107
classifying 168, 476	casinl 107
returning from input streams 300	complex argument subroutines
writing to streams 990	carg 107
charsetID	cargf 107
multibyte character 158	cargl 107
chdir subroutine 123	complex conjugate subroutines
chmod subroutine 124	conj 148
chown subroutine 127 chownx subroutine 127	conjf 148
chpass subroutine 129	conjl 148 complex cosine functions
chroot subroutine 129	conplex cosine functions ccos 114
chssys subroutine 132	ccosf 114
cimag subroutine 134	ccosl 114
cimag subroutine 134	complex exponential functions
cimagi subroutine 134	cexp 117
cjistosj subroutine 473	cexpf 117
ckuseracct subroutine 135	cexpl 117
ckuserID subroutine 137	complex hyperbolic cosine functions
class subroutine 138	ccosh 115
clearerr macro 227	ccoshf 115
clock subroutine 140	ccoshl 115
clog subroutine 140	complex hyperbolic sine subroutines
clogf subroutine 140	csinh 160
clogl subroutine 140	csinhf 160
close subroutine 141	csinhl 160
closedir subroutine 771	complex hyperbolic tangent subroutines
	ctanh 162

complex hyperbolic tangent subroutines (continued)	coredump subroutine 292
ctanhf 162	cos subroutine 153
ctanhl 162	cosf subroutine 153
complex imaginary functions	cosh subroutine 154
cimag 134	coshf subroutine 154
cimagf 134	coshl subroutine 154
cimagl 134	cosine subroutines
complex natural logarithm functions	acosf 17
clog 140	acosl 17
clogf 140	cosf 153
clogl 140	cosl 153
complex power subroutines	cosl subroutine 153
cpow 155	cpow subroutine 155
cpowf 155	cpowf subroutine 155
cpowl 155	cpowl subroutine 155
complex projection subroutines	cproj subroutine 156
cproj 156	cprojf subroutine 156
cprojf 156	cprojl subroutine 156
cprojl 156	creal subroutine 156
complex tangent functions	crealf subroutine 156
catan 109	creall subroutine 156
catanf 109	creat subroutine 763
catanl 109	Critical Section Subroutines
Complex tangent subroutines	BeginCriticalSection Subroutine 186
ctan 161	EnableCriticalSections Subroutine 186
ctanf 161	EndCriticalSection Subroutine 186
ctanl 161	crypt subroutine 157
Computes the base 2 exponential.	csid subroutine 158
exp2 207	csin subroutine 159
exp2f 207	csinf subroutine 159
exp2l 207	csinh subroutine 160
confstr subroutine 147	csinhf subroutine 160
conj subroutine 148	csinhl subroutine 160
conjf subroutine 148	csinl subroutine 159
conjl subroutine 148	csjtojis subroutine 473
controlling terminal 162	csjtouj subroutine 473
conv subroutines 149	csqrt subroutine 160
conversion	csqrtf subroutine 160
date and time representations 166	csgrtl subroutine 160
date and time to string representation	cstoccsid subroutine 115
using asctime subroutine 166	ctan subroutine 161
using ctime subroutine 166	ctanf subroutine 161
using gmtime subroutine 166	ctanh subroutine 162
using localtime subroutine 166	ctanhf subroutine 162
converter subroutines	ctanhl subroutine 162
btowc 103	ctanl subroutine 161
fwscanf 286	ctermid subroutine 162
iconv_close 429	ctime subroutine 163
iconv_open 430	ctime_r subroutine 166
icode 473	ctype subroutines 168
mbrlen 668	cube root functions
mbrtowc 669	cbrtf 113
mbsinit 673	cbrtl 113
mbsrtowcs 678	cujtojis subroutine 473
swscanf 286	cujtosj subroutine 473
wscanf 286	current process credentials
copysignf subroutine 151	reading 341
copysignl subroutine 151	current process environment
core files	reading 343
coredump subroutine 292	current processes
gencore subroutine 292	getting user name 170
-	- -

current processes (continued)	directory subroutines (continued)
group ID	telldir 771
initializing 456	dirname Subroutine 173
returning 325	disclaim subroutine 174
path name of controlling terminal 162 user ID	div subroutine 4
	dlclose subroutine 176 dlerror subroutine 177
returning 381 current working directory	dlopen Subroutine 178
getting path name 308	dlsym Subroutine 178
cursor positions	double precission numbers
setting 452	frexpf 271
cuserid subroutine 170	drand48 subroutine 181
cuseria subroduire 170	drem subroutine 183
	dup subroutine 214
D	dup2 subroutine 214
_	dupz subroduite 214
data arrays	
encrypting 157 data locks 816	E
	_
data sorting subroutines	ecvt subroutine 185
bsearch 102	EnableCriticalSections Subroutine 186
ftw 278	encrypt subroutine 157
hcreate 425	encryption
hdestroy 425	performing 157
hsearch 425	EndCriticalSection Subroutine 186
insque 457	endfsent subroutine 316
Ifind 644	endfsent_r subroutine 367
Isearch 644	endgrent subroutine 318
remque 457	endhostent subroutine 740
data space segments	endpwent subroutine 354
changing allocation 100	endrpcent subroutine 359
date	endttyent subroutine 379
displaying and setting 374	endutent subroutine 393
date format conversions 163	endvfsent subroutine 395
defect 219851 946	environment variables
defect 220239 349	finding default PATH 147
defssys subroutine 171 delssys subroutine 171	finding values 313 setting 992
	erand48 subroutine 181
descriptor tables getting size 313	erf subroutine 187
difftime subroutine 163	erfc subroutine 188
directories	erfcf subroutine 188
changing 123	erff subroutine 187
changing root 131	errlog subroutine 189
creating 685	errlog_close subroutine 191
directory stream operations 771	errlog_find Subroutines
generating path names 401	errlog_find_first 192
getting path name of current directory 308	errlog_find_next 192
directory subroutines	errlog_find_sequence 192
chdir 123	errlog_find_first Subroutine 192
chroot 131	errlog_find_next Subroutine 192
closedir 771	errlog_find_sequence Subroutine 192
getcwd 308	errlog_open Subroutine 194
getwd 398	errlog_set_direction Subroutine 195
glob 401	errlog_write Subroutine 195
globfree 404	errlogging Subroutines
link 610	errlog_close 191
mkdir 685	errlog_open 194
opendir 771	errlog_set_direction 195
readdir 771	errlog_write 195
rewinddir 771	error functions
seekdir 771	computing 187

error functions (continued)	fchown subroutine 127
erff 187	fchownx subroutine 127
error handling	fclear subroutine 211
math 665	fclose subroutine 213
returning information 623	fcntl subroutine 214
error logs	fcvt subroutine 185
closing 191	fdetach Subroutine 221
finding 192	fdim subroutine 222
opening 194	fdimf subroutine 222
setting direction 195	fdiml subroutine 222
writing 195	fdopen subroutine 245
writing to 189	feclearexcept subroutine 223
error messages	fegetenv subroutine 223
placing into program 73	fegetexceptflag subroutine 224
writing 814	fegetround subroutine 225
errorlogging subroutines	feholdexcept subroutine 225
errlog 189	feof macro 227
perror 814	feraiseexcept subroutine 228
euclidean distance functions	ferror macro 227
hypotf 426	fesetenv subroutine 223
hypotl 426	fesetexceptflag subroutine 224
Euclidean distance functions	fesetround subroutine 225
computing 426	fetch_and_add subroutine
exec subroutines 196	atomic access 229
execl subroutine 196	fetch_and_and subroutine
execle subroutine 196	atomic access 230
execlp subroutine 196	fetch_and_or subroutine
exect subroutine 196	atomic access 230
execution profiling	fetestexcept subroutine 231
after initialization 696	feupdateenv subroutine 231
using default data areas 703	ffinfo subroutine 232
using defined data areas 697	fflush subroutine 213
execv subroutine 196	ffs subroutine 97
execve subroutine 196	fgetc subroutine 300
execvp subroutine 196	fgetpos subroutine 273
exit subroutine 203	fgets subroutine 366
exp subroutine 205	fgetwc subroutine 397
exp2 subroutine 207	fgetws subroutine 399
exp2f subroutine 207	FIFO files
exp2l subroutine 207	creating 687
expf subroutine 205	file access permissions
expm1 subroutine 208	changing 119, 124
expm1f subroutine 208	file descriptors
expm1l subroutine 208	checking I/O status 861
exponential functions	closing associated files 141
computing 205	controlling 214
exponential subroutines	establishing connections 763
·	
expf 205	performing control functions 459
expm1f, 208	file names
expm1l 208	constructing unique 689
	file ownership
_	changing 127
F	file permissions
fabs subroutine 209	changing 119, 124
fabsf subroutine 209	file pointers
fabsl subroutine 209	moving read-write 646
faccessx subroutine 5	file subroutines
fattach Subroutine 209	access 5
fchacl subroutine 119	accessx 5
fchdir Subroutine 211	dup 214
fchmod subroutine 124	dup2 214

file subroutines (continued)	file-implementation characteristics 774
endutent 393	fileno macro 227
faccessx 5	files
fclear 211	binary 267
fontl 214	
	closing 141
ffinfo 232	creating 687
finfo 232	creating links 610
flock 628	creating space at pointer 211
flockfile 233	determining accessibility 5
fpathconf 774	establishing connections 763
·	
fsync 276	generating path names 401
fsync_range 276	getting name list 738
ftrylockfile 233	locking and unlocking 628
funlockfile 233	opening 763
getc_unlocked 302	opening streams 245
getchar_unlocked 302	reading 267
•	
getenv 313	reading asynchronously 33
getutent 393	repositioning pointers 273
getutid 393	revoking access 270
getutline 393	systems
lockf 628	getting information about 367
lockfx 628	writing asynchronously 43
lseek 646	writing binary 267
mkfifo 687	finfo subroutine 232
mknod 687	finite subroutine 138
mkstemp 689	finite testing
mktemp 689	isfinite 464
nlist 738	first-in-first-out files 687
nlist64 738	flags
pathconf 774	returning 337
pclose 797	floating point multiply-add
pipe 815	fma 237
popen 864	fmaf 237
putc_unlocked 302	fmal 237
·	
putchar_unlocked 302	floating point numbers
putenv 992	Idexpf 591
pututline 393	ldexpl 591
setutent 393	nextafterf 730
utmpname 393	nextafterl 730
file system subroutines	nexttoward 730
confstr 147	nexttowardf 730
endfsent 316	nexttowardl 730
endvfsent 395	floating-point absolute value functions
fscntl 271	computing 234
getfsent 316	floating-point environment
getfsfile 316	feholdexcept 225
getfsspec 316	feupdateenv 231
getfstype 316	floating-point environment variables
0 11	
getvfsbyflag 395	fegetenv, 223
getvfsbyname 395	fesetenv 223
getvfsbytype 395	floating-point exception
getvfsent 395	feraiseexcept 228
mntctl 694	fetestexcept 231
setfsent 316	floating-point exceptions 254, 258, 263
setvfsent 395	changing floating point status and control
file systems	register 260
controlling operations 271	feclearexcept 223
retrieving information 316	flags 252
returning mount status 694	querying process state 263
file trees	testing for occurrences 256, 257
	103ting 101 000dille110e3 230, 231
searching recursively 278	

floating-point number subroutines	fp_enable subroutine 251
fdim 222	fp_enable_all subroutine 251
fdimf 222	fp_flush_imprecise Subroutine 255
fdiml 222	fp_inexact subroutine 256
floating-point numbers	fp_invalid_op subroutine 256
converting to strings 185	fp_iop_convert subroutine 257
determining classifications 138	fp_iop_infdinf subroutine 257
fmax 238	fp_iop_infmzr subroutine 257
fmaxf 238	fp_iop_infsinf subroutine 257
fmaxl 238	fp_iop_invcmp subroutine 257
fminf 239	fp_iop_snan subroutine 257
fminl 239	fp_iop_sqrt subroutine 257
fmodf 239	fp_iop_vxsoft subroutine 257
manipulating 695	fp_iop_zrdzr subroutine 257
modff 695	fp_is_enabled subroutine 251
reading and setting rounding modes 259	fp_overflow subroutine 256
rounding 234	fp_raise_xcp subroutine 258
floating-point rounding subroutines	fp_read_flag subroutine 252
nearbyint 729	fp_read_rnd subroutine 259
nearbyintf 729	fp_set_flag subroutine 252
nearbyintl 729	fp_sh_info subroutine 260
floating-point status flags	fp_sh_set_stat subroutine 260
fegetexceptflag 224	fp_sh_trap_info subroutine 260
fesetexceptflag 224	fp_swap_flag subroutine 252
floating-point subroutines 254, 258, 260, 263, 265	fp_swap_rnd subroutine 259
fp_sh_info 260	fp_trap subroutine 263
fp_sh_trap_info 260	fp_trapstate subroutine 265
floating-point trap control 251	fp_underflow subroutine 256
flock subroutine 628	fpathconf subroutine 774
flockfile subroutine 233	fpclassify macro 266
floor functions	fprintf subroutine 869
floorf 234	fputc subroutine 990
floor subroutine 234	fputs subroutine 994
floorf subroutine 234	fputwo subrouting 995
floorl subroutine 234 fma subroutine 237	fputws subroutine 997 fread subroutine 267
fmaf subroutine 237	free subroutine 657
fmal subroutine 237	freopen subroutine 245
fmax subroutine 238	frevoke subroutine 270
fmaxf subroutine 238	frexp subroutine 271
fmaxl subroutine 238	frexpf subroutine 271
fmin subroutine 661	frexpl subroutine 271
fminf subroutine 239	fscntl subroutine 271
fminl subroutine 239	fseek subroutine 273
fmod subroutine 239	fsetpos subroutine 273
fmodf subroutine 239	fsync subroutine 276
fmodl subroutine 239	fsync_range subroutine 276
fmout subroutine 661	ftell subroutine 273
fmtmsg Subroutine 240	ftime subroutine 374
fnmatch subroutine 243	ftok subroutine 277
fopen subroutine 245	ftrylockfile subroutine 233
fork subroutine 248	ftw subroutine 278
formatted output	funlockfile subroutine 233
printing 869	fwide subroutine 281
fp_any_enable subroutine 251	fwprintf subroutine 281
fp_any_xcp subroutine 256	fwrite subroutine 267
fp_clr_flag subroutine 252	fwscanf subroutine 286
fp_cpusync subroutine 254	
fp_disable subroutine 251	
fp_disable_all subroutine 251	G
fp_divbyzero subroutine 256	gai_strerror subroutine 290

gamma functions computing natural logarithms 291 gamma subroutine 291 gcd subroutine 661 gcvt subroutine 185 gencore subroutine 292 get_malloc_log subroutine 294 get_malloc_log_live subroutine 294 get_speed subroutine 295 getargs Subroutine 296 getaudithostattr, IDtohost, hosttoID, nexthost or putaudithostattr subroutine 298 getauthdb subroutine 299 getc subroutine 300 getc_unlocked subroutine 302 getchar subroutine 300 getchar_unlocked subroutine 302 getconfattr subroutine 303 getcontext or setcontext Subroutine 307 getcwd subroutine 308 getdate Subroutine 310 getdtablesize subroutine getegid subroutine 317 getenv subroutine 313 geteuid subroutine 381 getevars Subroutine 314 getfsent subroutine 316 getfsent_r subroutine 367 getfsfile subroutine 316 getfsspec subroutine 316 getfsspec_r subroutine 367 getfstype subroutine 316 getfstype_r subroutine 367 getgid subroutine 317 getgidx subroutine 317 getgrent subroutine 318 getgrgid subroutine 318 getgrgid r subroutine 319 getgrnam subroutine 318 getgrnam_r subroutine 320 getgroupattr subroutine 322 getgroups subroutine 325 getgrpaclattr Subroutine 326 gethostent subroutine 739 getinterval subroutine 329 getitimer subroutine 329 getlogin subroutine 335 getlogin_r subroutine 336 getopt subroutine 337 getpagesize subroutine 339 getpass subroutine 340 getpcred subroutine 341 getpenv subroutine getpgid Subroutine getpgrp subroutine 345 getpid subroutine 345 getportattr Subroutine 345 getppid subroutine 345 getpri subroutine 348 getpriority subroutine 349

getpwent subroutine 354 getpwnam subroutine 354 getpwuid subroutine 354 getrlimit subroutine 356 getrlimit64 subroutine 356 getroleattr Subroutine 363 getrpcbyname subroutine 359 getrpcbynumber subroutine getrpcent subroutine 359 getrusage subroutine 360 getrusage64 subroutine 360 gets subroutine 366 getsfile r subroutine 367 getsid Subroutine 368 getssys subroutine 369 getsubopt Subroutine 370 getsubsvr subroutine 371 gettimeofday subroutine 374 gettimer subroutine 375 gettimerid subroutine 378 getttyent subroutine 379 getttynam subroutine 379 getuid subroutine 381 getuidx subroutine 381 getuinfo subroutine 382 getuserattr subroutine 382 GetUserAuths Subroutine 389 getuserpw subroutine 389 getusraclattr Subroutine 391 getutent subroutine 393 getutid subroutine 393 getutline subroutine 393 getvfsbyflag subroutine 395 getvfsbyname subroutine 395 getvfsbytype subroutine 395 getvfsent subroutine 395 getw subroutine 300 getwc subroutine 397 getwchar subroutine 397 getwd subroutine 398 getws subroutine 399 glob subroutine 401 globfree subroutine 404 gmtime subroutine 163 gmtime_r subroutine 166 grantpt subroutine 404

Н

hash tables
manipulating 425
HBA_CloseAdapter Subroutine 405
HBA_FreeLibrary Subroutine 406
HBA_GetAdapterAttributes Subroutine 407
HBA_GetAdapterName Subroutine 409
HBA_GetDiscoveredPortAttributes Subroutine 407
HBA_GetFcpTargetMapping Subroutine 411
HBA_GetNumberOfAdapters Subroutine 412
HBA_GetPersistentBinding Subroutine 410
HBA_GetPortAttributes Subroutine 407
HBA_GetPortAttributesByWWN Subroutine 407

getpw Subroutine 353

HBA_GetPortStatistics Subroutine 413 HBA_GetRNIDMgmtInfo Subroutine 414 HBA_GetVersion Subroutine 415	I/O low-level subroutines <i>(continued)</i> open 763 I/O requests
HBA_LoadLibrary Subroutine 415 HBA_OpenAdapter Subroutine 416	canceling 22
HBA RefreshInformation Subroutine 417	retrieving error status 25
HBA_SendCTPassThru Subroutine 417	retrieving return status 37
HBA_SendReadCapacity Subroutine 418	suspending 40
HBA_SendReportLUNs Subroutine 419	I/O stream macros
HBA_SendRNID Subroutine 421	clearerr 227
HBA_SendScsiInquiry Subroutine 422	feof 227
HBA_SetRNIDMgmtInfo Subroutine 423	ferror 227
hcreate subroutine 425	fileno 227
hdestroy subroutine 425	I/O stream subroutines
Host Bus Adapter API	fclose 213
HBA_CloseAdapter 405	fdopen 245
HBA_FreeLibrary 406	fflush 213
HBA_GetAdapterAttributes 407	fgetc 300
HBA_GetAdapterName 409	fgetpos 273
HBA_GetDiscoveredPortAttributes 407	fgets 366
HBA_GetFcpPersistentBinding 410	fgetwc 397
HBA_GetFcpTargetMapping 411	fgetws 399
HBA_GetNumberOfAdapters 412	fopen 245
HBA_GetPortAttributes 407	fprintf 869
HBA_GetPortAttributesByWWN 407	fputc 990
HBA_GetPortStatistics 413	fputs 994
HBA_GetRNIDMgmtInfo 414	fputwc 995
HBA_GetVersion 415	fputws 997
HBA_LoadLibrary 415	fread 267
HBA_OpenAdapter 416	freopen 245
HBA_RefreshInformation 417	fseek 273
HBA_SendCTPassThru 417	fsetpos 273
HBA_SendReadCapacity 418	ftell 273
HBA_SendReportLUNs 419	fwide 281
HBA_SendRNID 421	fwprintf 281
HBA_SendScsilnquiry 422	fwrite 267
HBA_SetRNIDMgmtInfo 423	getc 300
hsearch subroutine 425	getchar 300
hyperbolic cosine subroutines coshf 154	gets 366 getw 300
coshl 154	getwc 397
hypot subroutine 426	getwc 397 getwchar 397
hypotf subroutine 426	getwentar 657 getws 399
hypotl subroutine 426	printf 869
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,	putc 990
_	putchar 990
1	puts 994
I/O asynchronous subroutines	putw 990
aio_cancel 22	putwc 995
aio_error 25	putwchar 995
aio_fsync 28	putws 997
aio_nwait 29	rewind 273
aio_nwait_timeout 31	sprintf 869
aio_read 33	swprintf 281
aio_return 37	vfprintf 869
aio_suspend 40	vprintf 869
aio_write 43	vsprintf 869
lio_listio 611	vwsprintf 869
poll 861	wprintf 281 wsprintf 869
I/O low-level subroutines 141, 763 creat 763	wapiiiti 00a

I/O terminal subroutines	IMTextCursor callback subroutine 452
cfsetispeed 118	IMTextDraw callback subroutine 453
ioctl 459	IMTextHide callback subroutine 453
ioctl32 459	IMTextStart callback subroutine 454
ioctl32x 459	imul_dbl subroutine 4
ioctlx 459	incinterval subroutine 329
iconv_close subroutine 429	inet_aton subroutine 455
iconv_open subroutine 430	infinity values
identification subroutines	isinf 466
endgrent 318	initgroups subroutine 456
endpwent 354	initialize subroutine 456
getconfattr 303	input method
getgrent 318	checking language support 450
getgrgid 318	closing 438
getgrnam 318	control and query operations 445
getgroupattr 322	creating instance 439
getpwent 354	destroying instance 439
getpwnam 354	initializing for particular language 443
getpwuid 354	input method keymap
getuinfo 382	initializing 441, 444
getuserattr 303, 382	mapping key and state pair to string 434, 447, 451
IDtogroup 322	input method subroutines
IDtouser 382	callback functions
nextgroup 322	IMAuxCreate 435
nextuser 382	IMAuxDestroy 435
putgroupattr 322	IMAuxDraw 436
putpwent 354	IMAuxHide 437
putuserattr 382	IMBeep 437
setgrent 318	IMIndicatorDraw 441
setpwent 354	IMIndicatorHide 442
idpthreadsa 157	IMTextCursor 452
IDtogroup subroutine 322	IMTextDraw 453
IDtouser subroutine 382	IMTextHide 453
IEE Remainders	IMTextStart 454
computing 183	IMAIXMapping 434
ilogb subroutine 432	IMClose 438
ilogbf subroutine 432	IMCreate 439
ilogbl subroutine 432	IMDestroy 439
IMAIXMapping subroutine 434	IMFilter 440
IMAuxCreate callback subroutine 435	IMFreeKeymap 441
IMAuxDestroy callback subroutine 435	IMinitialize 443
IMAuxDraw callback subroutine 436	IMInitializeKeymap 444
IMAuxHide callback subroutine 437	IMIoctl 445
imaxabs subroutine 433	IMLookupString 447
imaxdiv subroutine 433	IMProcess 448
IMBeep callback subroutine 437	IMProcessAuxiliary 449
IMClose subroutine 438	IMQueryLanguage 450
IMCreate subroutine 439	IMSimpleMapping 451
IMDestroy subroutine 439	input streams
IMFilter subroutine 440	reading character string from 399
IMFreeKeymap subroutine 441	reading single character from 397
IMIndicatorDraw callback subroutine 441	returning characters or words 300
IMIndicatorHide callback subroutine 442	insque subroutine 457
IMInitialize subroutine 443	install_lwcf_handler() subroutine 458
IMInitializeKeymap subroutine 444	integers
IMloctl subroutine 445	computing absolute values 4
IMLookupString subroutine 447	computing division 4
IMProcess subroutine 448	computing double-precision multiplication 4
IMProcessAuxiliary subroutine 449	performing arithmetic 661
IMQueryLanguage subroutine 450	Internet addresses
IMSimpleMapping subroutine 451	converting to ASCII strings 455

interoperability subroutines	iswupper subroutine 470
ccsidtocs 115	iswxdigit subroutine 470
cstoccsid 115	isxdigit subroutine 168
interprocess channels	itom subroutine 661
creating 815	itrunc subroutine 234
interprocess communication keys 277	
interval timers	
allocating per process 378	J
manipulating expiration time 329	i0 subroutine 98
returning values 329	j1 subroutine 98
inverse hyperbolic cosine subroutines	Japanese conv subroutines 474
acoshf 18	Japanese ctype subroutines 474
acoshl 18	icode subroutines 473
inverse hyperbolic functions	JFS
computing 18, 71	controlling operations 271
inverse hyperbolic sine subroutines	JIS character conversions 473
asinhf 71	iistoa subroutine 474
asinhl 71	jistosj subroutine 473
inverse hyperbolic tangent subroutines	jistouj subroutine 473
atanhf 75	in subroutine 98
atanhl 75	Journaled File System 214
invert subroutine 661	jrand48 subroutine 181
ioctl subroutine 459	jianu46 Subroutine 161
ioctl32 subroutine 459	
ioctl32x subroutine 459	K
ioctlx subroutine 459	
is_wctype subroutine 472	Kanji character conversions 473
isalnum subroutine 168	keyboard events
isalpha subroutine 168	processing 440, 448
isascii subroutine 168	kill subroutine 478
isblank subroutine 462	killpg subroutine 478
iscntrl subroutine 168	kleenup subroutine 479
isdigit subroutine 168	knlist subroutine 480
isendwin Subroutine 463	kpidstate subroutine 482
isfinite macro 464	kutentojis subroutine 474
isgraph subroutine 168	
isgreater macro 464	1
isgreaterequal subroutine 465	L
isinf subroutine 466	l3tol subroutine 484
isless macro 466	l64a subroutine 1
islessequal macro 467	labs subroutine 4
islessgreater macro 468	LAPI_Addr_get subroutine 485
islower subroutine 168	LAPI_Addr_set subroutine 486
isnan subroutine 138	LAPI_Address subroutine 488
isnormal macro 468	LAPI_Address_init subroutine 489
isprint subroutine 168	LAPI_Address_init64 491
ispunct subroutine 168	LAPI_Amsend subroutine 492
isspace subroutine 168	LAPI_Amsendv subroutine 498
isunordered macro 469	LAPI_Fence subroutine 506
	LAPI_Get subroutine 507
isupper subroutine 168 iswalnum subroutine 470	LAPI_Getcntr subroutine 509
	LAPI_Getv subroutine 511
	LAPI_Gfence subroutine 514
iswblank subroutine 471 iswcntrl subroutine 470	LAPI_Init subroutine 515
	LAPI_Msg_string subroutine 521
iswetype subroutine 472	LAPI_Msgpoll subroutine 522
iswdigit subroutine 470	LAPI_Nopoll_wait subroutine 524
iswgraph subroutine 470	LAPI_Probe subroutine 526
iswlower subroutine 470	LAPI_Purge_totask subroutine 527
iswprint subroutine 470	LAPI_Put subroutine 528
iswpunct subroutine 470	LAPI_Putv subroutine 530
iswspace subroutine 470	LAPI Qenv subroutine 534

LAPI_Resume_totask subroutine	538	Ilroundl subroutine 617
LAPI_Rmw subroutine 539		load subroutine 618
LAPI_Rmw64 subroutine 543		loadbind subroutine 622
LAPI_Senv subroutine 546		loadquery subroutine 623
LAPI_Setcntr subroutine 548		locale subroutines
LAPI_Setcntr_wstatus subroutine	550	localeconv 625
LAPI_Term subroutine 552		nl_langinfo 736
LAPI_Util subroutine 553		locale-dependent conventions 625
LAPI_Waitcntr subroutine 561		localeconv subroutine 625
LAPI_Xfer structure types 564 LAPI_Xfer subroutine 563		locales returning language information 736
lapi_xfer_type_t 564		localtime subroutine 163
layout values		localtime r subroutine 166
querying 580		lockf subroutine 628
setting 582		lockfx subroutine 628
transforming text 585		log gamma functions
LayoutObject		Igamma 608
creating 576		Igammaf 608
freeing 588		lgammal 608
Icong48 subroutine 181		log subroutine 635
Idaclose subroutine 590		log10 subroutine 631
Idahread subroutine 589		log10f subroutine 631
Idaopen subroutine 599		log1p subroutine 632
Idclose subroutine 590		log1pf subroutine 632
Idexp subroutine 591		log1pl subroutine 632
Idexpf subroutine 591		log2 subroutine 633
Idexpl subroutine 591		log2f subroutine 633
Idfhread subroutine 592		log2l subroutine 633
Idgetname subroutine 594		logarithmic functions
Idiv subroutine 4		computing 205
Idlinit subroutine 596		logb subroutine 634
Idlitem subroutine 596		logbf subroutine 634
IdInseek subroutine 597		logbl subroutine 634
Idlread subroutine 596		logf subroutine 635
Idlseek subroutine 597 Idnrseek subroutine 601		logical volumes
Idnshread subroutine 602		querying 647 login name
Idnsseek subroutine 604		getting 335, 336
Idohseek subroutine 598		loginfailed Subroutine 636
Idopen subroutine 599		loginrestrictions Subroutine 638
Idrseek subroutine 601		loginsuccess Subroutine 641
Idshread subroutine 602		long integers
Idsseek subroutine 604		converting to strings 2
Idtbindex subroutine 605		long integers, converting
Idtbread subroutine 606		to 3-byte integers 484
Idtbseek subroutine 607		to base-64 ASCII strings 1
Ifind subroutine 644		Irand48 subroutine 181
Igamma subroutine 608		Irint subroutine 642
Igammaf subroutine 608		Irintf subroutine 642
Igammal subroutine 608		Irintl subroutine 642
linear searches 644		Iround subroutine 643
lineout subroutine 609		Iroundf subroutine 643
link subroutine 610		Iroundl subroutine 643
lio_listio subroutine 611		Isearch subroutine 644
llabs subroutine 4		Iseek subroutine 646
Ildiv subroutine 4		Itol3 subroutine 484
Ilrint subroutine 616		LVM logical volume subroutines
Ilrintf subroutine 616 Ilrintl subroutine 616		lvm_querylv 647
Ilround subroutine 617		LVM physical volume subroutines lvm quervpv 651
mound addividuine OT/		IVIII UUGIVUV UJI

Ilroundf subroutine 617

LVM volume group subroutines lvm_queryvg 654 lvm_queryvgs 656 lvm_querylv subroutine 647 lvm_querypv subroutine 651 lvm_queryvg subroutine 654 lvm_queryvgs subroutine 656	memory management (continued) defining addresses 184 defining available paging space 878 disclaiming memory content 174 generating IPC keys 277 returning system page size 339 memory management subroutines alloca 657
	calloc 657
M	disclaim 174
m in subroutine 661	free 657
m out subroutine 661	ftok 277
macros	gai_strerror 290
assert 73	getpagesize 339
madd subroutine 661	madvise 663
madvise subroutine 663	mallinfo 657
makecontext Subroutine 664	mallinfo_heap 657
mallinfo subroutine 657	malloc 657
mallinfo_heap subroutine 657	mallopt 657
malloc subroutine 657	memccpy 683
mallopt subroutine 657	memchr 683
mapped files	memcmp 683
synchronizing 723	memcpy 683
MatchAllAuths Subroutine 667	memmove 683
MatchAllAuthsList Subroutine 667	memset 683 mincore 684
MatchAnyAuthsList Subroutine 667	
math errors	mmap 690 moncontrol 696
handling 665	monitor 697
matherr subroutine 665 mblen subroutine 667	monstartup 703
mbrlen subroutine 668	mprotect 706
mbrtowc subroutine 669	msem_init 707
mbsadvance subroutine 670	msem_lock 709
mbscat subroutine 671	msem_remove 710
mbschr subroutine 672	msem_unlock 711
mbscmp subroutine 671	msleep 722
mbscpy subroutine 671	msync 723
mbsinit subroutine 673	munmap 726
mbsinvalid subroutine 674	mwakeup 727
mbslen subroutine 674	psdanger 878
mbsncat subroutine 675	realloc 657
mbsncmp subroutine 675	memory mapping
mbsncpy subroutine 675	advising system of paging behavior 663
mbspbrk subroutine 676	determining page residency status 684
mbsrchr subroutine 677	file-system objects 690
mbsrtowcs subroutine 678	modifying access protections 706
mbstomb subroutine 679	putting a process to sleep 722 semaphores
mbstowcs subroutine 680	initializing 707
mbswidth subroutine 681	locking 709
mbtowc subroutine 681	removing 710
mcmp subroutine 661 mdiv subroutine 661	unlocking 711
memccpy subroutine 683	synchronizing mapped files 723
memchr subroutine 683	unmapping regions 726
memcmp subroutine 683	waking a process 727
memcpy subroutine 683	memory pages
memmove subroutine 683	determining residency 684
memory allocation 657	memory semaphores
memory area operations 683	initializing 707
memory management	locking 709
controlling execution profiling 696, 697, 703	putting a process to sleep 722

memory semaphores (continued) removing 710	multibyte character subroutines <i>(continued)</i> mbscat 671
unlocking 711	mbschr 672
waking a process 727	mbscmp 671
memset subroutine 683	mbscpy 671
message catalogs	mbsinvalid 674
closing 110	mbslen 674
opening 112	mbsncat 675
retrieving messages 111	mbsncmp 675
message control operations 712	mbsncpy 675
message facility subroutines	mbspbrk 676
catclose 110	mbsrchr 677
catgets 111	mbstomb 679
catopen 112	mbstowcs 680
message queue identifiers 714	mbswidth 681
message queues	mbtowc 681
checking I/O status 861	multibyte characters
reading messages from 715	converting to wide 680, 681
receiving messages from 720	determining display width of 681
sending messages to 718	determining length of 667
min subroutine 661	determining number of 674
mincore subroutine 684	extracting from string 679
mkdir subroutine 685	locating character sequences 676
mkfifo subroutine 687	locating next character 670
mknod subroutine 687	locating single characters 672, 677
mkstemp subroutine 689	operations on null-terminated strings 671, 675
mktemp subroutine 689	returning charsetID 158
mktime subroutine 163	validating 674
mmap subroutine 690	munmap subroutine 726
mntctl subroutine 694	mwakeup subroutine 727
modf subroutine 695	invarioup dubloutino 121
modff subroutine 695	
modfl subroutine 695	N
modulo remainders	
computing 234	NaN
moncontrol subroutine 696	nan 728
monitor subroutine 697	nanf 728
monstartup subroutine 703	nanl 728
mout subroutine 661	nan subroutine 728
move subroutine 661	nanf subroutine 728
mprotect subroutine 706	nanl subroutine 728
mrand48 subroutine 181	natural logarithm functions
msem_init subroutine 707	logf 635
msem_lock subroutine 709	logi 635
msem_remove subroutine 710	natural logarithms
msem_unlock subroutine 711	log1pf 632
msgctl subroutine 712	log1pl 632
msgget subroutine 714	NCesc subroutine 149
msgrcv subroutine 715	NCflatchr subroutine 149
msgsnd subroutine 718	NCtolower subroutine 149
msgxrcv subroutine 720	NCtoNLchar subroutine 149
msleep subroutine 722	NCtoupper subroutine 149
-	NCunesc subroutine 149
msqrt subroutine 661 msub subroutine 661	nearbyint subroutine 729
msync subroutine 723	nearbyintf subroutine 729
-	nearbyintl subroutine 729
mt_trce() subroutine 724	nearest subroutine 234
multibute character subroutines	network host entries
multibyte character subroutines	retrieving 739
csid 158	new-process image file 196
mblen 667	newpass subroutine 732
mbsadvance 670	nextafter subroutine 730

nextafterf subroutine 730	numerical manipulation subroutines (continued)
nextafterl subroutine 730	fabs 209
nextgroup subroutine 322	fabsl 209
nextgrpacl Subroutine 326	fcvt 185
nextrole Subroutine 363	finite 138
nextrore Subroutine 303	floor 234
nexttowardf subroutine 730	floorl 234
nexttowardl subroutine 730	fmin 661
nextuser subroutine 382	fmod 239
nextusracl Subroutine 391	fmodl 239
nftw subroutine 734	fp_any_enable 251
nice subroutine 349	fp_any_xcp 256
nl_langinfo subroutine 736	fp_clr_flag 252
nlist subroutine 738	fp_disable 251
nlist64 subroutine 738	fp_disable_all 251
nrand48 subroutine 181	fp_divbyzero 256
number manipulation function	fp_enable 251
copysignf 151	fp_enable_all 251
copysigni 151	fp_inexact 256
numbers	• —
	fp_invalid_op 256
generating	fp_iop_convert 257
pseudo-random 181	fp_iop_infdinf 257
numerical manipulation subroutines 291	fp_iop_infmzr 257
a64l 1	fp_iop_infsinf 257
abs 4	fp_iop_invcmp 257
acos 17	fp_iop_snan 257
acosf 17	fp_iop_sqrt 257
acosh 18	fp_iop_zrdzr 257
acosl 17	fp_is_enabled 251
asin 72	fp_overflow 256
asinh 71	fp_read_flag 252
asini 72	fp_read_rnd 259
atan 75	fp_set_flag 252
atan2 73	· · · · · · · · · · · · · · · · · · ·
	fp_swap_flag 252
atan2f 73	fp_swap_rnd 259
atan2l 73	fp_underflow 256
atanf 75	frexp 271
atanh 75	frexpl 271
atanhf 75	gamma 291
atanhl 75	gcd 661
atanl 75	gcvt 185
atof 77	hypot 426
atoff 77	ilogb 432
atol 78	imul_dbl 4
atoll 78	invert 661
cabs 426	isnan 138
cbrt 113	itom 661
ceil 116	itrunc 234
ceilf 116	j0 98
ceill 116	ju 36 j1 98
	•
class 138	jn 98
cos 153	jrand48 181
div 4	l3tol 484
drand48 181	l64a 1
drem 183	labs 4
ecvt 185	lcong48 181
erand48 181	ldexp 591
erf 187	ldexpl 591
erfc 188	ldiv 4
exp 205	llabs 4
expm1 208	Ildiv 4

numerical manipulation subroutines (continued)	object file access subroutines (continued)
log 635	ldshread 602
log10 631	ldsseek 604
log1p 632	ldtbindex 605
logb 634	ldtbread 606
Irand48 181	ldtbseek 607
Itol3 484	object file subroutines
m_in 661	load 618
m_out 661	loadbind 622
madd 661	loadquery 623
matherr 665	object files
mcmp 661	closing 590
mdiv 661	computing symbol table entries 605
min 661	controlling run-time resolution 622
modf 695	listing 623
modfl 695	loading and binding 618
mout 661	manipulating line number entries 596
move 661	providing access 599
mrand48 181	reading archive headers 589
msqrt 661	reading file headers 592
msub 661	reading indexed section headers 602
mult 661	reading symbol table entries 606
nearest 234	retrieving symbol names 594
nextafter 730	seeking to indexed sections 604
nrand48 181	seeking to line number entries 597
omin 661	seeking to optional file header 598
omout 661	seeking to relocation entries 601
pow 661, 866	seeking to symbol tables 607
rpow 661	objects
sdiv 661	setting locale-dependent conventions 625
seed48 181	ODM
srand48 181	ending session 762
trunc 234	error message strings 745
uitrunc 234	freeing memory 746
umul_dbl 4	ODM (Object Data Manager)
unordered 138	initializing 751
y0 98	running specified method 759
y1 98	ODM object classes
yn 98	adding objects 740
,	changing objects 742
	closing 743
0	creating 744
Object Data Manager 751	locking 752
object file access subroutines	opening 755
Idaclose 590	removing 757
Idahread 589	removing objects 756, 758
Idaopen 599	retrieving class symbol structures 753
Idelose 590	retrieving objects 747, 748, 750
Idfhread 592	setting default path location 760
Idgetname 594	setting default permissions 761
Idlinit 596	unlocking 763
Idlitem 596	ODM subroutines
Idiread 596	odm_add_obj 740
	odm_change_obj 742
ldlseek 597 Idnlseek 597	odm_close_class 743
	odm_create_class 744
Idnrseek 601	odm_create_class 744 odm_err_msg 745
Idnshread 602	odm_free_list 746
Idnsseek 604	odm_nee_ist 740 odm_get_by_id 747
Idohseek 598	odm_get_by_id 747
Idopen 599	odm_get_list 730
ldrseek 601	ouiii_get_iist 140

ODM subroutines (continued)	passwords
odm_get_next 750	generating new 732
odm_get_obj 750	reading 340
odm_initialize 751	pathconf subroutine 774
odm_lock 752	pause subroutine 777
odm_mount_class 753	pcap_close 778
odm_open_class 755	pcap_compile 779
odm_rm_by_id 756	pcap_datalink 779
odm_rm_class 757	pcap_dispatch 780
odm_rm_obj 758	pcap_dump 781
odm_run_method 759	pcap_dump_close 782
odm_set_path 760	pcap_dump_open 783
odm_set_perms 761	pcap_file 783
odm_terminate 762	pcap_fileno 784
odm_unlock 763	pcap_geterr 785
odm_add_obj subroutine 740	pcap_is_swapped 785
odm_change_obj subroutine 742	pcap_lookupdev 786
odm_close_class subroutine 743	pcap_lookupnet 787
odm_create_class subroutine 744	pcap_loop 788
odm_err_msg subroutine 745	pcap_major_version 789
odm_free_list subroutine 746	pcap_next 790
odm_get_by_id subroutine 747	pcap_open_live 791
odm_get_first subroutine 750	pcap_open_offline 792
odm_get_list subroutine 748	pcap_perror 793
odm_get_next subroutine 750	pcap_setfilter 794
odm_get_obj subroutine 750	pcap_snapshot 795
odm initialize subroutine 751	pcap_stats 795
odm_lock subroutine 752	pcap_strerror 796
odm_mount_class subroutine 753	pclose subroutine 797
odm_open_class subroutine 755	perfstat_cpu subroutine 798
odm_rm_by_id subroutine 756	perfstat_cpu_total subroutine 799
odm_rm_class subroutine 757	perfstat_disk subroutine 801
odm_rm_obj subroutine 758	perfstat_disk_total subroutine 805
odm_run_method subroutine 759	perfstat_diskadapter subroutine 802
odm_set_path subroutine 760	perfstat_diskpath subroutine 803
odm_set_perms subroutine 761	perfstat_memory_total subroutine 806
odm terminate subroutine 762	perfstat_netbuffer subroutine 807
odm unlock subroutine 763	perfstat_netinterface subroutine 808
omin subroutine 661	perfstat_netinterface_total subroutine 810
omout subroutine 661	perfstat_pagingspace subroutine 811
open file descriptors	perfstat_protocol subroutine 812
controlling 214	perfstat_reset subroutine 814
performing control functions 459	permanent storage
open subroutine	writing file changes to 276
described 763	perror subroutine 814
opendir subroutine 771	pglob parameter
openx subroutine	freeing memory 404
described 763	physical volumes
output stream	querying 651
writing character string to 997	pipe subroutine 815
writing single character to 995	pipes
withing single character to 995	closing 797
	creating 815, 864
P	plock subroutine 816
-	poll subroutine 861
paging memory	popen subroutine 864
behavior 663	
defining available space 878	posix_openpt Subroutine 865
passwdexpired 773	pow subroutine 661, 866
password maintenance	power functions
password changing 129	computing 205
	powf 866

powf subroutine 866	process subroutines (security and auditing) (continued)
powl subroutine 866	getgroups 325
pre-editing space 454	getpcred 341
print formatter subroutines	getpenv 343
initialize 456	getuid 381
lineout 609	getuidx 381
print lines	initgroups 456
formatting 609	kleenup 479
printer initialization 456	process user IDs
printf subroutine 869	returning 381
process accounting	processes
displaying resource use 360	closing pipes 797
enabling and disabling 8	creating 248
tracing process execution 978	getting process table entries 351
process credentials	initializing run-time environment 479
reading 341	initiating pipes 864
process environments	suspending 777
initializing run-time 479	terminating 3, 203, 478
reading 343	tracing 978
process group IDs	processes subroutines
returning 317, 345	_exit 203
supplementary IDs	abort 3
getting 325	acct 8
initializing 456	atexit 203
process identification	brk 100
alphanumeric user name 170	ctermid 162
path name of controlling terminal 162	cuserid 170
process IDs	exec 196
returning 345	exit 203
process initiation	fork 248
creating child process 248	getdtablesize 313
executing file 196	getpgrp 345
process locks 816	getpid 345
process messages	getppid 345
getting message queue identifiers 714	getpri 348
providing control operations 712	getpriority 349
reading from message queue 715	getrlimit 356
receiving from message queue 720	getrlimit64 356
sending to message queue 718	getrusage 360
process priorities	getrusage64 360
getting or setting 349	kill 478
returning scheduled priorities 348	killpg 478
process program counters	msgctl 712
histogram 876	msgget 714
process resource allocation	msgrcv 715
changing data space segments 100	msgsnd 718
controlling system consumption 356	msgxrcv 720
getting size of descriptor table 313	nice 349
locking into memory 816	pause 777
starting address sampling 876	plock 816
stopping address sampling 876	profil 876
process resource use 360	psignal 879
process signals	ptrace 978
alarm 329	sbrk 100
printing system signal messages 879	setpriority 349
sending to processes 478	setrlimit 356
The state of the s	
process subroutines (security and auditing)	setrlimit64 356
getegid 317	times 360 vfork 248
geteuid 381	
getgid 317	vlimit 356
getgidx 317	vtimes 360

profil subroutine 876	pthread_mutex_destroy subroutine 951
program assertion	pthread_mutex_init subroutine 951
verifying 73	PTHREAD_MUTEX_INITIALIZER macro 952
psdanger subroutine 878	pthread_mutex_lock subroutine 953
psignal subroutine 879	pthread_mutex_trylock subroutine 953
pthdb_attr_	pthread_mutexattr_destroy subroutine 955
pthdb_attr_addr 881	pthread_mutexattr_getkind_np subroutine 956
pthdb_attr_detachstate 881	pthread_mutexattr_gettype subroutine 957
pthdb_attr_guardsize 881	pthread_mutexattr_init subroutine 955
pthdb_attr_inheritsched 881	pthread_mutexattr_setkind_np subroutine 959
pthdb_attr_schedparam 881	pthread_mutexattr_settype subroutine 957
pthdb_attr_schedpolicy 881	pthread_once subroutine 961
pthdb_attr_schedpriority 881	PTHREAD_ONCE_INIT macro 962
pthdb_attr_scope 881	pthread_self subroutine 971
pthdb_attr_stackaddr 881	pthread_setcancelstate subroutine 972
pthdb_attr_stacksize 881	pthread_setschedparam subroutine 973
pthdb_attr_suspendstate 881	pthread_setspecific subroutine 941
pthread_atfork subroutine 905	pthread_signal_to_cancel_np subroutine 975
pthread_attr_destroy subroutine 906	pthread_suspend_np and pthread_continue_np
pthread_attr_getdetachstate subroutine 913	subroutine 976
pthread_attr_getguardsize subroutine 907	pthread_unlock_global_np subroutine 977
pthread_attr_getschedparam subroutine 908	pthread_yield subroutine 978
pthread_attr_getscope subroutine 914	ptrace subroutine 978
pthread_attr_getstackaddr subroutine 909	ptracex subroutine 978
pthread_attr_getstacksize subroutine 910	ptsname subroutine 989
pthread_attr_init subroutine 911	putc subroutine 990
pthread_attr_setdetachstate subroutine 913	putc_unlocked subroutine 302
pthread_attr_setguardsize subroutine 907	putchar subroutine 990
pthread_attr_setschedparam subroutine 915	putchar_unlocked subroutine 302
pthread_attr_setscope subroutine 914	putenv subroutine 992
pthread_attr_setstackaddr subroutine 916	putgrent subroutine 993
pthread_attr_setstacksize subroutine 917	putgroupattr subroutine 322
pthread_attr_setsupendstate_np and	putgrpaclattr Subroutine 326
pthread_attr_getsuspendstate_np subroutine 918	putportattr Subroutine 345
pthread_cancel subroutine 919	putpwent subroutine 354
pthread_cleanup_pop subroutine 920	putroleattr Subroutine 363
pthread_cleanup_push subroutine 920	puts subroutine 994
pthread_cond_broadcast subroutine 924	putuserattr subroutine 382
pthread_cond_destroy subroutine 921	putuserpw subroutine 389
PTHREAD_COND_INITIALIZER macro 923	putuserpwhist subroutine 389
pthread_cond_signal subroutine 924	putusraclattr Subroutine 391
pthread_cond_timedwait subroutine 925	pututline subroutine 393
pthread_cond_wait subroutine 925	putw subroutine 990
pthread_condattr_destroy subroutine 927	putwo subroutine 995
pthread_condattr_getpshared subroutine 928	putwchar subroutine 995
pthread_condattr_setpshared subroutine 929	putws subroutine 997
pthread_create subroutine 930	
pthread_delay_np subroutine 932 pthread_equal subroutine 933	Q
pthread_exit subroutine 934	
pthread_get_expiration_np subroutine 935	queues
pthread_getconcurrency subroutine 936	inserting and removing elements 457
pthread_getrusage_np subroutine 937	quotient and remainder
pthread_getschedparam subroutine 940	imaxdiv 433
pthread_getspecific subroutine 941	
pthread_getunique_np subroutine 945	R
pthread_join subroutine 946	
pthread_key_create subroutine 947	radix-independent exponents
pthread_key_delete subroutine 948	logbf 634
pthread_kill subroutine 949	logbl 634
pthread_lock_global_np subroutine 950	read operations
p a.a_ioon_giobai_np oabioaiiio ooo	asynchronous 33

read operations (continued)	setrlimit64 subroutine 356
binary files 267	setrpcent subroutine 359
read-write file pointers	setsockopt subroutine 427
moving 646	settimeofday subroutine 374
readdir subroutine 771	settimer subroutine 375
real floating types	setttyent subroutine 379
fpclassify 266	setutent subroutine 393
real value subroutines	setvfsent subroutine 395
creal 156	shell command-line flags 337
crealf 156	SIGALRM signal 330
creall 156	SIGIOT signal 3
realloc subroutine 657	signal names
regular expressions	formatting 879
matching patterns 144	sine functions
remque subroutine 457	csin 159
resabs subroutine 329	csinf 159
reset_speed subroutine 295	csinl 159
resinc subroutine 329	single-byte to wide-character conversion 103
resource information 937	SJIS character conversions 473
resources subroutines	sitojis subroutine 473
pthread_getrusage_np 937 restimer subroutine 375	sitouj subroutine 473
rewind subroutine 273	snprintf subroutine 869
	socket options
rewinddir subroutine 771	setting 427
rounding direction	sockets kernel service subroutines
fegetround 225	setsockopt 427
fesetround 225	sockets network library subroutines
rounding numbers	endhostent 740
Ilrint 616	gethostent 739
Ilrintf 616	inet_aton 455
Ilrintl 616	special files
llround 617	creating 687
llroundf 617	sprintf subroutine 869
Ilroundl 617	square root subroutines
Irint 642	csqrt 160
Irintf 642	csqrtf 160
Irintl 642	csqrtl 160
Iround 643	srand48 subroutine 181
Iroundf 643	SRC subroutines
Iroundl 643	addssys 19
rpc file	chssys 132
handling 359	delssys 171
rpow subroutine 661	getssys 369
run-time environment	SRC subsys record
initializing 479	adding 19
	SRC subsys structure
	initializing 171
S	Statistics subroutines
sbrk subroutine 100	perfstat_cpu 798
sdiv subroutine 661	perfstat_cpu_total 799
seed48 subroutine 181	perfstat_disk 801
seekdir subroutine 771	perfstat_disk_total 805
set_speed subroutine 295	perfstat_diskadapter 802
setfsent subroutine 316	perfstat_diskpath 803
setfsent_r subroutine 367	perfstat_memory_total 806
setgrent subroutine 318	perfstat_netbuffer 807
setitimer subroutine 329	perfstat_netinterface 808
setkey subroutine 157	perfstat_netinterface_total 810
	perfstat_pagingspace 811
setpriority subroutine 349	perfetat_protocol 812
setpwent subroutine 354	peristat_protocol 612 peristat_reset 814
setrlimit subroutine 356	Pootat_1000t 011

status indicators	subroutines
beeping 437	LAPI_Addr_get 485
drawing 441	LAPI_Addr_set 486
hiding 442	LAPI_Address 488
step subroutine 144	LAPI_Address_init 489
stime subroutine 375	LAPI_Address_init64 491
streams	LAPI_Amsend 492
checking status 227	LAPI_Amsendv 498
closing 213	LAPI_Fence 506
flushing 213	LAPI_Get 507
opening 245	LAPI_Getcntr 509
repositioning file pointers 273	LAPI_Getv 511
writing to 213	LAPI_Gfence 514
string conversion	LAPI_Init 515
long integers to base-64 ASCII 1	LAPI_Msg_string 521
string manipulation subroutines	LAPI_Msgpoll 522
advance 144	LAPI_Nopoll_wait 524
bcmp 97	LAPI_Probe 526
bcopy 97	LAPI_Purge_totask 527
bzero 97	LAPI_Put 528
compile 144	LAPI_Putv 530
ffs 97	LAPI_Qenv 534
fgets 366	LAPI_Resume_totask 538
fnmatch 243	LAPI_Rmw 539
fputs 994	LAPI_Rmw64 543
gets 366	LAPI_Senv 546
puts 994	—
step 144	LAPI_Setcntr 548 LAPI_Setcntr_wstatus 550
· · · · · · · · · · · · · · · · · · ·	LAPI_Selcitit_Wstatus 550 LAPI_Term 552
strings	
bit string operations 97	LAPI_Util 553
byte string operations 97	LAPI_Waitcntr 561
copying 97	LAPI_Xfer 563
drawing text strings 453	Subroutines
matching against pattern parameters 243	perfstat_cpu 798
reading bytes into arrays 366	perfstat_cpu_total 799
writing to standard output streams 994	perfstat_disk_total 801, 805
zeroing out 97	perfstat_diskpath 803
subroutine	perfstat_memory_total 806
pcap_close 778	perfstat_netinterface_total 808, 810
pcap_compile 779	subsystem objects
pcap_datalink 779	modifying 132
pcap_dispatch 780	removing 171
pcap_dump 781	subsystem records
pcap_dump_close 782	reading 369, 371
pcap_dump_open 783	supplementary process group IDs
pcap_file 783	getting 325
pcap_fileno 784	initializing 456
pcap_geterr 785	swapcontext Subroutine 664
pcap_is_swapped 785	swprintf subroutine 281
pcap_lookupdev 786	swscanf subroutine 286
pcap_lookupnet 787	symbol-handling subroutine
pcap_loop 788	knlist 480
pcap_major_version 789	symbols
pcap_next 790	translating names to addresses 480
pcap_open_live 791	sys_siglist vector 879
pcap_open_offline 792	system auditing 79
pcap_perror 793	system data objects
pcap_setfilter 794	auditing modes 86
pcap_snapshot 795	system event audits
pcap_stats 795	getting or setting status 83
pcap_strerror 796	

system resources	Threads Library (continued)
setting maximums 356	thread-specific data
system signal messages 879	pthread_getspecific subroutine 941
system variables	pthread_key_create subroutine 947
determining values 147	pthread_key_delete subroutine 948
	pthread_setspecific subroutine 941 threads
Т	cancellation 919, 972
telldir subroutine 771	creation 930
terminal baud rate	creation attributes 906, 909, 910, 911, 913, 914
get 295	916, 917, 918, 976
set 295	ID handling 933, 971
text area	initialization 961, 962
hiding 453	termination 920, 934, 946
text locks 816	time
text strings	displaying and setting 374
drawing 453	reporting used CPU time 140
Thread-safe C Library	synchronizing system clocks 21
subroutines	time format conversions 163
164_r 2	time manipulation subroutines
Thread-Safe C Library 319, 320, 367	absinterval 329 adjtime 21
subroutines	alarm 329
getfsent_r 367	asctime 163
getlogin_r 336 getsfile_r 367	clock 140
setfsent_r 367	ctime 163
threads	difftime 163
getting thread table entries 372	ftime 374
Threads Library 973	getinterval 329
condition variables	getitimer 329
creation and destruction 921, 923	gettimeofday 374
creation attributes 927, 928, 929	gettimer 375
signalling a condition 924	gettimerid 378
waiting for a condition 925	gmtime 163
DCE compatibility subroutines	incinterval 329
pthread_delay_np 932	localtime 163
pthread_get_expiration_np 935	mktime 163
pthread_getunique_np 945	resabs 329 resinc 329
pthread_lock_global_np 950	restimer 375
pthread_mutexattr_getkind_np 956 pthread_mutexattr_setkind_np 959	setitimer 329
pthread_indiexatti_setkind_rip 959 pthread_signal_to_cancel_np 975	settimeofday 374
pthread_unlock_global_np 977	settimer 375
mutexes	stime 375
creation and destruction 952	time 375
creation attributes 957	tzset 163
locking 953	ualarm 329
pthread_mutexattr_destroy 955	time subroutine 375
pthread_mutexattr_init 955	timer
process creation	getting or setting values 375
pthread_atfork subroutine 905	times subroutine 360
pthread_attr_getguardsize subroutine 907	toascii subroutine 149
pthread_attr_setguardsize subroutine 907	tojhira subroutine 474
pthread_getconcurrency subroutine 936	tojkata subroutine 474 tojlower subroutine 474
pthread_mutex_destroy 951	tojupper subroutine 474
pthread_mutex_init 951	tolower subroutine 149
scheduling dynamic thread control 940, 978	toujis subroutine 474
thread creation attributes 908, 915	toupper subroutine 149
signal, sleep, and timer handling	trace
pthread_kill subroutine 949	install_lwcf_handler subroutine 458

trace (continued)	vtimes subroutine 360
mt_trce subroutine 724	vwsprintf subroutine 869
transforming text 585	·
trunc subroutine 234	
trusted processes	W
initializing run-time environment 479	wide character subroutines
tty description file	fgetwc 397
querying 379	fgetws 399
tty subroutines	fputwo 995
endttyent 379	fputws 997
getttyent 379	getwo 397
getttynam 379	getwo 397 getwchar 397
setttyent 379	getws 399
tzset subroutine 163	is_wctype 472
	iswalnum 470
	iswalpha 470
U	iswentrl 470
ualarm subroutine 329	iswctype subroutine 472
uitrunc subroutine 234	
	iswdigit 470
UJIS character conversions 473	iswgraph 470 iswlower 470
ujtojis subroutine 473	
ujtosj subroutine 473	iswprint 470
umul_dbl subroutine 4	iswpunct 470
unbiased exponents	iswspace 470
ilogbf 432	iswupper 470
ilogbl 432	iswxdigit 470
unordered subroutine 138	putwo 995
user accounts	putwchar 995
checking validity 135	putws 997
user authentication data	wide characters
accessing 389	checking character class 470
user database	converting
accessing group information 318, 322	from multibyte 680, 681
accessing user information 303, 354, 382	determining properties 472
user information	reading from input stream 397, 399
accessing 303, 354, 382	writing to output stream 995, 997
accessing group information 318, 322	words
searching buffer 382	returning from input streams 300
user login name	wprintf subroutine 281
getting 335	write operations
users	asynchronous 43
authenticating 137	binary files 267
utmpname subroutine 393	wscanf subroutine 286
	wsprintf subroutine 869
V	
V	Υ
vectors	ĭ
sys_siglist 879	y0 subroutine 98
vfork subroutine 248	y1 subroutine 98
vfprintf subroutine 869	yn subroutine 98
VFS (Virtual File System)	
getting file entries 395	
returning mount status 694	
virtual memory	
mapping file-system objects 690	
vlimit subroutine 356	
volume groups	
querying 654	
querying all varied on-line 656	
vprintf subroutine 869	
vsprintf subroutine 869	

Readers' Comments — We'd Like to Hear from You

Phone No.

AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions, Volume 1

Publication No. SC23-41	59-06				
Overall, how satisfied are you with the information in this book?					
Overall satisfaction	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
How satisfied are you th	at the information	in this book is:			
Accurate Complete Easy to find Easy to understand Well organized Applicable to your tasks	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Please tell us how we ca	an improve this bo	ok:			
Thank you for your respon	s to IBM, you grant	IBM a nonexclus	ive right to use or	distribute your co	omments in any
way it believes appropriate	e without incurring a	any obligation to s	vou.		
Name		Ade	dress		
Company or Organization					

Readers' Comments — We'd Like to Hear from You SC23-4159-06



Cut or Fold Along Line

Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation Information Development Department H6DS-905-6C006 11501 Burnet Road Austin, TX 78758-3493



Hadlaldadddalalalaldaldaallalal

Fold and Tape

Please do not staple

Fold and Tape

IBM

Printed in the U.S.A.

SC23-4159-06

