

IBM

**OS Open
Programmer's Reference**

Ninth Edition (September 1997)

This edition of *IBM OS Open Programmer's Reference* applies to IBM OS Open Version 1.6.1 and to subsequent versions of the OS Open Real-Time Operating System until otherwise indicated in new versions or technical newsletters.

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS MANUAL "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND 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.

IBM does not warrant that the contents of this publication or the accompanying source code examples, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying source code examples are error-free.

This publication could contain 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 program(s) described in this publication at any time.

It is possible that this publication may contain references to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Any reference to an IBM licensed program in this publication is not intended to state or imply that you can use only IBM's licensed program. You can use any functionally equivalent program instead.

No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the written permission of IBM.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Address comments about this publication to:

IBM Corporation
Department H83A
P.O. Box 12195
Research Triangle Park, NC 27709

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

©Copyright International Business Machines Corporation 1993, 1997. All rights reserved.

Printed in the United States of America.

4 3 2 1

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

Patents, Trademarks, and Acknowledgments

IBM may have patents or pending patent applications covering the subject matter in this publication. The furnishing of this publication does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 208 Harbor Drive, Stamford, CT 06904, United States of America.

The following terms, denoted by an asterisk (*) at their first appearance in this publication, are trademarks of IBM Corporation.

AIX	POWER Architecture
IBM	PowerPC Architecture
OS Open	RISC System/6000

The following term, denoted by a double asterisk (**) at its first appearance in this publication, is a trademark of UNIX System Laboratories, Inc.

UNIX

The OS Open software and documentation are based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following institution for its role in the development: the Electrical Engineering and Computer Sciences Department at the Berkeley Campus.

Portions of the code and documentation described in this book were derived from code and documentation developed under the auspices of the Regents of the University of California and have been acquired and modified under the provisions that the following copyright notice and permission notice appear:

©Copyright Regents of the University of California, 1986, 1987. All rights reserved.

Redistribution and use in source and binary forms are permitted provided that this notice is preserved and that due credit is given the University of California at Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. This software is provided "as is" without express or implied warranty.

Some of the C language functions described herein use function names and prototypes derived from the following standards and draft standards from the American National Standards Institute (ANSI) and the Institute of Electrical and Electronic Engineers (IEEE):

- ANSI X3.159-1989 (ANSI C)
- IEEE Std 1003.1b-1993 (POSIX)
- IEEE Draft Standard 1003.1c/D9
- IEEE Draft Standard 1003.13/D5

Contents

Tables	xxv
About This Book	xxvii
Who Should Use This Book	xxvii
How to Use This Book.....	xxvii
How to Read the Syntax Diagrams in This Book	xxviii
Conventions Used in this Book	xxix
Numeric Notation and Input Conventions	xxix
Highlighting Conventions	xxix
Product Support	xxx
Where to Find More Information	xxx
Related IBM Publications	xxx
Standards.....	xxx
 Chapter 1. OS Open Function and Macro Summaries.....	 1
System Management Functions.....	2
Interrupt Handling Functions.....	2
Signal Handling Functions	3
Thread Management Functions	4
Thread Attributes Handling Functions.....	5
Thread Condition Handling Functions.....	7
Thread Condition Attribute Handling Functions	7
Thread Group Handling Functions	8
Rate Monotonic Scheduler Functions	9
Message Queue Handling Functions	9
Mutex Handling Functions.....	10
Mutex Attributes Handling Functions	10
Semaphore Handling Functions.....	11
Shared Memory Handling Functions.....	12
Clock and Timer Functions	12
Memory Allocation Functions	13
Virtual Memory Management Functions	14
Memory Map Management Functions.....	15
Memory Heap Management Functions	15
Memory Pool Management Functions	15
Device Management Functions.....	16
PCMCIA Handling Functions	17
TTY Handling Functions.....	17
File Handling Functions.....	18
Root Directory Entry Management Functions (Virtual Memory Only)	21
Input/Output Functions.....	21
TCP/IP Networking and Network Management Functions.....	23
Network File System (NFS) Functions	27
Remote Procedure Call Support Functions.....	27
Loader Functions	33
Symbol Table Functions.....	34

Character Handling Utilities.....	35
Math Utilities.....	36
Queue Utilities.....	38
Ring Buffer Utilities.....	39
String Handling Utilities	39
General Programming Utilities	41
Debugging and Reporting Functions.....	43
OpenShell Functions.....	46
Chapter 2. OS Open Function and Macro Descriptions	47
Attributes and Threads.....	47
Async Safe Functions.....	47
Cancel Safe Functions	48
Interrupt Handler Safe Functions	49
Character Sets Used in Function and Macro Descriptions.....	49
Decimal Digits	49
Graphics Characters	49
Hexadecimal Digits.....	49
Lowercase Letters	49
Uppercase Letters	49
Portable File Name Characters	49
abort().....	50
abs()	51
accept()	52
access().....	55
acos()	57
alarm().....	58
align_h()	60
asctime()	62
asctime_r()	64
asin()	66
assert()	67
atan().....	68
atan2().....	69
atexit()	70
atof()	71
atoi()	72
atol()	74
auth_destroy(), AUTH_DESTROY().....	76
authnone_create().....	77
authunix_create()	78
authunix_create_default()	79
bind()	80
bootp_request().....	82
bprintf().....	84

bprintf_set()	85
bsearch()	86
_callxlc()	88
calloc()	89
callrpc()	90
ceil()	91
cfgetispeed()	92
cfgetospeed()	93
cfsetispeed()	94
cfsetospeed()	95
chmod()	97
clearDisp()	98
clearerr()	99
cLib_init()	100
clnt_broadcast()	101
clnt_call(), CLNT_CALL()	103
clnt_control(), CLNT_CONTROL()	104
clnt_create()	106
clnt_destroy(), CLNT_DESTROY()	107
clnt_freeres(), CLNT_FREERES()	108
clnt_geterr(), CLNT_GETERR()	109
clnt_pcreateerror()	110
clnt_perrno()	111
clnt_perror()	112
clnt_spccreateerror()	113
clnt_sperrno()	114
clnt_sperror()	115
clntraw_create()	116
clnttcp_create()	117
clntudp_bufcreate()	119
clntudp_create()	121
clock()	123
clock_getres()	124
clock_gettime()	125
clock_settime()	126
close()	127
closedir()	128
cond_dump()	129
cond_list()	130
connect()	132
cos()	134
cosh()	135
cpp_exit()	136

cpp_init()	137
cs_card_status()	138
cs_get_io_window()	139
cs_get_mem_window()	140
cs_init()	141
cs_int_install()	143
cs_int_query()	144
cs_pata_parm_init()	145
cs_set_ccr_offset()	147
cs_stateless_card_status()	148
ctime()	149
ctime_r()	150
dbbrkpt_clr()	151
dbbrkpt_cont()	152
dbbrkpt_disp()	153
dbbrkpt_find_inst()	154
dbbrkpt_init()	155
dbbrkpt_isbp()	156
dbbrkpt_next_inst()	157
dbbrkpt_set()	159
dbbrkpt_tclr()	160
dbbrkpt_tset()	161
dbbrkpt_write_inst()	162
dbmem_disp()	163
dbmem_listi()	164
dbstepi()	165
dbsymname_find()	166
dbwhere()	167
dcache_flush()	168
dcache_invalidate()	169
decompress()	170
dev_io_init()	171
device_install()	172
device_uninstall()	174
difftime()	175
div()	176
driver_install()	177
dtom()	178
enet_arpinput()	179
enet_arpresolve()	181
enet_attach()	183
errlog()	185
exit()	187

exp()	188
fabs()	189
fclose()	190
fcntl()	191
fdopen()	193
feof()	195
ferror()	196
fflush()	197
fgetc()	198
fgetpos()	199
fgets()	201
fileno()	202
flih_list()	203
flockfile()	205
floor()	208
fmod()	209
fopen()	210
format()	215
fpathconf()	217
fpreg_dump()	219
fprintf()	221
fputc()	222
fputs()	224
fread()	225
free()	228
freopen()	229
frexp()	230
fs_init()	231
fscanf()	234
fseek()	235
fsetpos()	237
fstat()	238
ftell()	239
ftp()	241
ftpd_start()	249
ftruncate()	250
ftrylockfile()	251
funlockfile()	252
fwrite()	253
get_myaddress()	254
getc()	255
getc_unlocked()	256
getchar()	257

getchar_unlocked()	258
getcwd()	261
getenv()	262
gethostbyaddr()	263
gethostbyname()	265
gethostlock()	266
gethostname()	267
gethostunlock()	268
getnetbyaddr()	270
getnetbyname()	272
getnetlock()	273
getnetunlock()	274
getopt()	275
getpeername()	277
getpid()	278
getprompt()	279
getprotobyname()	280
getprotobynumber()	281
getprotolock()	282
getprotounlock()	283
getrpcbyname()	285
getrpcbynumber()	286
getrpccent()	287
getrpcport()	288
gets()	289
getservbyname()	290
getservbyport()	291
getservlock()	293
getservunlock()	294
getsockname()	295
getsockopt()	296
gmtime()	299
gmtime_r()	301
heap_list()	302
if_attach()	303
ifconfig()	304
inbyte()	307
inet_addr()	308
inet_aton()	309
inet_lnaof()	311
inet_makeaddr()	312
inet_netof()	313
inet_network()	314

inet_ntoa()	316
inet_ntoa_r()	317
info_rtable()	318
inshort()	319
int_disable()	320
int_enable()	321
int_flihdisable()	322
int_flihenable()	324
int_flihregister()	325
int_getid()	326
int_install()	327
int_query()	329
inword()	331
ioctl()	332
isalnum()	333
isalpha()	335
isatty()	336
iscntrl()	337
isdigit()	338
isgraph()	339
islower()	340
isprint()	341
ispthreadid()	342
ispunct()	343
isspace()	344
issuspended()	345
istgid()	346
isupper()	347
isxdigit()	348
kda_dump()	349
kill()	350
labs()	351
ld()	352
ldexp()	354
ldiv()	355
ldrDup_name()	356
ldrEntry_get()	359
ldrIsexec()	360
ldrLink()	361
ldrLoad()	363
ldrMoveData()	366
ldrMoveSym()	367
ldrMsg()	368

ldrQimage()	369
ldrResolve()	370
ldrSym_get()	371
ldrUnlink()	373
ldr_ldinfo()	374
ldrFree()	377
ldrLib_init()	378
ldrQuery()	379
ldrStatus()	380
ldrUnload()	381
library_list()	382
link()	384
listen()	386
localeconv()	387
localtime()	388
localtime_r()	389
log()	390
log10()	391
longjmp()	392
lseek()	393
m_free()	394
m_freem()	395
M_PREPEND()	396
m_prepend()	397
malloc()	398
mblen()	399
mbtowc()	400
mbstowcs()	401
memchr()	402
memcmp()	403
memcpy()	404
memheap_alloc()	405
memheap_alloc_aligned()	406
memheap_alloc_pages()	407
memheap_extend()	408
memheap_free()	409
memheap_query()	410
memheap_realloc()	413
memheap_replace()	414
memmove()	415
mempool_alloc()	417
mempool_destroy()	418
mempool_free()	419

mempool_init()	420
mempool_query()	422
memset()	423
MFREE()	424
MGET()	425
MGETHDR()	426
mkdir()	427
mktime()	429
mlock()	430
mlockall()	431
mmap()	432
modf()	434
mq_close()	435
mq_dump()	436
mq_getattr()	437
mq_list()	438
mq_notify()	439
mq_open()	441
mq_receive()	446
mq_send()	449
mq_setattr()	452
mq_timedreceive()	453
mq_timedsend()	455
mq_unlink()	457
mtod()	458
munlock()	459
munlockall()	460
munmap()	461
mutex_dump()	462
mutex_list()	463
mutexattr_dump()	465
nanosleep()	466
net_init()	468
net_splimp()	469
net_splx()	470
nfs_authget()	471
nfs_authset()	472
nfs_dinit()	473
nfs_mount()	474
nfs_showmount()	475
nfs_umount()	476
np_compare_swap()	477
open()	478

opendir()	480
outbyte()	481
outshort()	482
outword()	483
package_install()	484
pata_init()	485
pathconf()	486
perror()	488
ping()	489
pmap_getmaps()	491
pmap_getport()	492
pmap_rmtcall()	493
pmap_set()	495
pmap_unset()	496
pool_list()	497
portmap_thread()	498
pow()	499
printf()	500
pthread_attr_destroy()	506
pthread_attr_getallocstack_np()	507
pthread_attr_getdetachstate()	508
pthread_attr_getfp_np()	509
pthread_attr_getinheritsched()	510
pthread_attr_getpr_np()	511
pthread_attr_getschedparam()	512
pthread_attr_getschedpolicy()	513
pthread_attr_getscope()	514
pthread_attr_getstackaddr()	515
pthread_attr_getstacksize()	516
pthread_attr_gettg_np()	517
pthread_attr_gettgid_np()	518
pthread_attr_init()	519
pthread_attr_setallocstack_np()	520
pthread_attr_setdetachstate()	521
pthread_attr_setfp_np()	522
pthread_attr_setinheritsched()	523
pthread_attr_setschedparam()	524
pthread_attr_setschedpolicy()	526
pthread_attr_setscope()	527
pthread_attr_setstackaddr()	528
pthread_attr_setstacksize()	529
pthread_attr_settg_np()	530
pthread_attr_settgid_np()	532

pthread_cancel()	533
pthread_cleanup_pop()	534
pthread_cleanup_push()	535
pthread_cond_broadcast()	538
pthread_cond_destroy()	539
pthread_cond_init()	540
pthread_cond_signal()	543
pthread_cond_timedwait()	544
pthread_cond_wait()	546
pthread_condattr_destroy()	548
pthread_condattr_getpshared()	549
pthread_condattr_init()	550
pthread_condattr_setpshared()	551
pthread_create()	552
pthread_detach()	554
pthread_equal()	555
pthread_errno_np()	556
pthread_exit()	557
pthread_getpr_np()	559
pthread_getschedparam()	560
pthread_getspecific()	561
pthread_getstackaddr_np()	562
pthread_join()	563
pthread_key_create()	564
pthread_key_delete()	567
pthread_kill()	568
pthread_mutex_destroy()	569
pthread_mutex_getprioceiling()	570
pthread_mutex_init()	571
pthread_mutex_lock()	572
pthread_mutex_setprioceiling()	573
pthread_mutex_timedlock()	574
pthread_mutex_trylock()	575
pthread_mutex_unlock()	576
pthread_mutexattr_destroy()	577
pthread_mutexattr_getprioceiling()	578
pthread_mutexattr_getprotocol()	579
pthread_mutexattr_getpshared()	580
pthread_mutexattr_init()	581
pthread_mutexattr_setprioceiling()	584
pthread_mutexattr_setprotocol()	585
pthread_mutexattr_setpshared()	586
pthread_once()	587

pthread_resume_np()	588
pthread_self()	589
pthread_setcancelstate()	590
pthread_setcanceltype()	591
pthread_setschedparam()	592
pthread_setspecific()	593
pthread_sigmask()	594
pthread_suspend_np()	596
pthread_testcancel()	597
pthread_tgcreate_np()	598
pthread_tgdestroy_np()	599
pthread_tggetkernel_np()	600
pthread_tggetspecific_np()	601
pthread_tgkeycreate_np()	603
pthread_tgkeydelete_np()	605
pthread_tgself_np()	606
pthread_tgsetspecific_np()	607
pthread_whattg_np()	608
putc()	609
putc_unlocked()	611
putchar()	612
putchar_unlocked()	613
puts()	615
qsort()	616
queCreate()	617
queDeq()	619
queDeqNoCheck()	620
queEnq()	621
queEnqFront()	622
queInit()	623
queInsert()	624
queNewer()	625
queNewest()	626
queOlder()	627
queOldest()	628
queRemove()	629
raise()	630
rand()	631
rand_r()	632
read()	633
readdir()	636
readdir_r()	637
realloc()	638

recv()	640
recvfrom()	643
recvmsg()	645
reg_get()	647
reg_set()	648
registerrpc()	649
rename()	650
rewind()	652
rewinddir()	653
rmdir()	654
rms_delq()	655
rms_init()	656
rms_insq()	657
rms_oldest()	658
rms_start()	659
rngBufGet()	662
rngBufPut()	663
rngCount()	664
rngCreate()	665
rngDelete()	666
rngFlush()	667
rngIsEmpty()	668
rngIsFull()	669
rootname_add()	670
rootname_delete()	671
rootname_get()	672
rootname_query()	673
route()	674
rpc_thread_init()	676
rsld_setschedparam()	677
rsld_start()	678
run()	679
scanf()	681
sched_getmask()	687
sched_setmask()	688
sched_yield()	689
schednetisr()	690
select()	691
select_notify()	694
select_redrive()	695
sem_close()	696
sem_destroy()	697
sem_getvalue()	698

sem_init()	699
sem_open()	700
sem_post()	705
sem_timedwait()	706
sem_trywait()	707
sem_unlink()	708
sem_wait()	709
semaphore_dump()	710
semaphore_list()	711
send()	713
sendmsg()	716
sendto()	718
setbuf()	720
setenv_np()	721
setjmp()	722
setlocale()	723
setprompt()	724
setsockopt()	725
setsysconf()	728
setuname()	729
setvbuf()	730
shell()	732
shell_threadid()	733
shm_open()	734
shm_unlink()	738
shutdown()	739
sigaddset()	740
sigdelset()	741
sigemptyset()	742
sigfillset()	743
sigismember()	744
siglongjmp()	745
signal()	746
signal_list()	747
sigpending()	748
sigsetjmp()	750
sigwait()	751
sin()	753
sinh()	754
sleep()	755
slip_attach()	756
slip_resume()	758
slip_suspend()	759

socket()	760
socket_services()	762
sprintf()	763
sqrt()	764
srand()	765
ss_init()	766
sscanf()	768
stat()	770
statsym_find_addr()	771
statsym_find_ext()	772
statsym_find_ldsym()	773
statsym_find_name()	775
statsym_find_tgaddr()	776
statsym_find_tgname()	777
statsym_find_tgtoc()	778
statsym_find_toc()	779
statsym_init()	780
statsym_remove()	781
statsym_update()	782
strategy()	783
strcasecmp()	785
strcat()	786
strchr()	787
strcmp()	790
strcoll()	791
strcpy()	792
strcspn()	793
strerror()	794
strftime()	795
strlen()	797
strncasecmp()	798
strncat()	799
strncmp()	800
strncpy()	801
strpbrk()	802
strrchr()	803
strspn()	804
strstr()	805
strtod()	806
strtok()	808
strtok_r()	811
strtol()	813
strtoul()	815

strxfrm()	817
svc_destroy(), SVC_DESTROY()	818
svc_freeargs(), SVC_FREEARGS()	819
svc_getargs(), SVC_GETARGS()	820
svc_getcaller(), SVC_GETCALLER()	821
svc_getreq()	822
svc_getreqset()	823
svc_register()	824
svc_run()	825
svc_sendreply()	826
svc_unregister()	827
svcerr_auth()	828
svcerr_decode()	829
svcerr_noproc()	830
svcerr_noprog()	831
svcerr_progvers()	832
svcerr_systemerr()	833
svcerr_weakauth()	834
svcfid_create()	835
svccraw_create()	836
svctcp_create()	837
svcudp_bufcreate()	838
svcudp_create()	839
symptr_get()	840
sysconf()	841
sysconfig()	844
system()	846
tan()	847
tanh()	848
tcdrain()	849
tcflow()	850
tcflush()	852
tcgetattr()	854
tcgetkey()	856
tcpip_init()	857
tcsendbreak()	858
tcsetattr()	860
tcsetkey()	862
telnetd_assignfunc()	863
telnetd_assigntty()	865
telnetd_start()	866
tftp()	870
tftp_init()	873

tg_dump()	874
tg_dump_all()	876
tg_list()	878
thread_attr_dump()	879
thread_debug()	880
thread_dump()	881
thread_info_list()	883
thread_list()	884
thread_name()	885
time()	886
time_request()	887
timer_create()	888
timer_delete()	890
timer_dump()	891
timer_getoverrun()	892
timer_gettime()	893
timer_list()	895
timer_settime()	897
timertick_notify()	899
tmpfile()	900
tmpnam()	901
tn()	902
tok_arpinut()	904
tok_arpresolve()	905
tolower()	906
toupper()	907
trace_checkall()	908
trace_checklast()	909
trace_copy()	910
trace_get()	911
trace_snapshot()	912
trace_write()	913
tty_init()	915
tzset()	918
uname()	920
ungetc()	921
unlink()	923
utime()	924
va_arg()	926
va_end()	928
va_start()	929
vfprintf()	930
vm_addrrealpages()	932

vm_allocate()	935
vm_copy()	937
vm_deallocate()	939
vm_protect()	941
vm_query()	943
vm_read()	945
vm_region()	947
vm_remove_realpages()	949
vm_translate()	951
vm_write()	953
vprintf()	955
vsprintf()	956
wctomb()	958
wcstombs()	959
write()	960
xdr_accepted_reply()	962
xdr_array()	963
xdr_authunix_params()	964
xdr_bool()	965
xdr_bytes()	966
xdr_callhdr()	967
xdr_callmsg()	968
xdr_char()	969
xdr_destroy(), XDR_DESTROY()	970
xdr_double()	971
xdr_enum()	972
xdr_float()	973
xdr_free()	974
xdr_getpos(), XDR_GETPOS()	975
xdr_inline(), XDR_INLINE()	976
xdr_int()	977
xdr_long()	978
xdr_opaque()	979
xdr_opaque_auth()	980
xdr_pmap()	981
xdr_pmaplist()	982
xdr_pointer()	983
xdr_reference()	984
xdr_rejected_reply()	985
xdr_replymsg()	986
xdr_setpos(), XDR_SETPOS()	987
xdr_short()	988
xdr_string()	989

xdr_u_char()	990
xdr_u_int()	991
xdr_u_long()	992
xdr_u_short()	993
xdr_union()	994
xdr_vector()	995
xdr_void()	996
xdr_wrapstring()	997
xdrmem_create()	998
xdrrec_create()	999
xdrrec_endofrecord()	1000
xdrrec_eof()	1001
xdrrec_skiprecord()	1002
xdrstdio_create()	1003
xprt_register()	1004
xprt_unregister()	1005
Index	1007

Tables

Table 1. System Management Functions	2
Table 2. Interrupt Handling Functions	2
Table 3. Signal Handling Functions	3
Table 4. Thread Management Functions	4
Table 5. Thread Attributes Handling Functions	5
Table 6. Thread Condition Handling Functions	7
Table 7. Thread Condition Handling Functions	7
Table 8. Thread Group Handling Functions	8
Table 9. Rate Monotonic Scheduler Functions	9
Table 10. Message Queue Handling Functions	9
Table 11. Mutex Handling Functions	10
Table 12. Mutex Attributes Handling Functions	10
Table 13. Semaphore Handling Functions	11
Table 14. Shared Memory Handling Functions	12
Table 15. Clock and Timer Functions	12
Table 16. Memory Allocation Functions	13
Table 17. Virtual Memory Management Functions	14
Table 18. Memory Map Management Functions	15
Table 19. Memory Heap Management Functions	15
Table 20. Memory Pool Management Functions	15
Table 21. Device Management Functions	16
Table 22. PCMCIA Handling Functions	17
Table 23. TTY Handling Functions	17
Table 24. File Handling Functions	18
Table 25. Root Directory Entry Management Functions (Virtual Memory Only)	21
Table 26. Input/Output Functions	21
Table 27. TCP/IP Networking and Network Management Functions	23
Table 28. Network File System (NFS) Functions	27
Table 29. Remote Procedure Call Support Functions	27
Table 30. Loader Functions	33
Table 31. Symbol Table Functions	34
Table 32. Character Handling Utilities	35
Table 33. Math Utilities	36
Table 34. Queue Utilities	38
Table 35. Ring Buffer Utilities	39
Table 36. String Handling Utilities	39

Table 37. General Programming Utilities 41

Table 38. Debugging and Reporting Functions 43

Table 39. OpenShell Functions 46

Table 40. Functions Not Async Safe 48

About This Book

This book provides a reference for the OS Open* real-time operating system. Although some information about using the OS Open system is provided, this book is not intended as a user's guide.

The OS Open operating system features:

- Hard real-time support, including deterministic execution and priority inversion protocols
- Board support packages for plug-and-play operation of popular board-level products
- Conformance with existing American National Standards Institute (ANSI) C and Portable Operating System Interface (POSIX) standards for application portability, with support for draft POSIX standards for real-time systems
- Open network interfaces to support embedded systems in heterogeneous environments
- Scalable implementations to meet the requirements and constraints of a variety of embedded systems

Who Should Use This Book

This book is for programmers who develop programs for embedded systems and understand:

- Their host systems' functions and features
- PowerPC Architecture* and assembler programming
- C programming using ANSI C and POSIX libraries

How to Use This Book

This book describes OS Open features and functions and provides reference descriptions of OS Open functions and macros in the OS Open libraries.





Chapter 1, "OS Open Function and Macro Summaries," contains tables of summary function and macro descriptions sorted within categories.

Chapter 2, "OS Open Function and Macro Descriptions," contains complete descriptions of the functions and macros, sorted alphabetically. Each description includes a code synopsis, library information, functional description, and error and attributes information. Where appropriate, descriptions contain examples and references.

An index of function names follows the function and macro descriptions.

How to Read the Syntax Diagrams in This Book

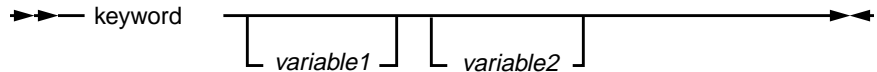
Throughout this book, diagrams illustrate the syntax for string formats and commands. The following list shows how to read these diagrams:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.
- A  symbol begins a diagram.
- A  symbol indicates continuation of a diagram on the next line.
- A  symbol indicates continuation of a diagram from the previous line.
- A  symbol terminates a diagram.
- Keywords are in regular type, and variables are in italics. Keywords must be typed exactly as shown.

- Keywords or variables on the main path of a diagram are required.

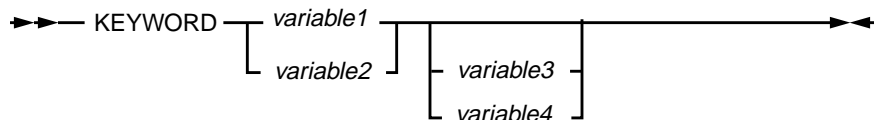


- Keywords or variables shown on branches below the main path are optional.

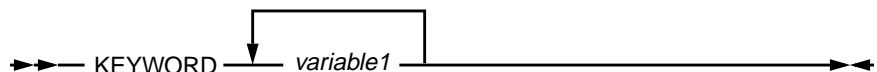


- Keywords or variables can appear in a stack, indicating that only one item in a stack can be chosen. If an item in a stack is on the main path, you must choose an item from the stack. If all items in a stack are below the main path, you may choose an item from the stack.

For example, in the following syntax diagram, you must choose either *variable1* or *variable2*. However, because *variable3* and *variable4* are below the main path, neither is required.



- A repeat separator is a returning arrow that surrounds a syntax element or group and shows that the element or group can be repeated.



Conventions Used in this Book

This book uses numeric and highlighting notation conventions based on those used in the RISC System/6000* and AIX* publications.

Numeric Notation and Input Conventions

In general, numbers are used exactly as shown. Unless noted otherwise, all numbers are in decimal, and, if entered as part of a command, are entered without format information.

In text, hexadecimal numbers are preceded by X followed by the number enclosed in single quotes, for example:

X'1A7'

In code examples and messages, hexadecimal numbers are preceded by "0x" or "x" followed by the number, which may be enclosed in single quotes, for example:

0x1a7, x'd1a7'

In text, the hexadecimal digits A through F appear in uppercase. In programs, these digits are typically entered in lowercase.

Highlighting Conventions

This book uses the following highlighting conventions:

- The names of invariant objects known to the OS Open operating system appear in bold type. In some text, however, such as in lists, no special typographic treatment is used. Examples of such objects include:
 - Function and macro names
 - Data types and structures
 - Constants and flags

Names of objects known to the OS Open operating system must be entered exactly as shown.

- Variable names supplied by user programs appear in italic type. In some text, however, such as in lists, no special typographic treatment is used. Examples of these objects include arguments and other parameters.
- No highlighting appears in code examples.

Product Support

If you have questions or encounter problems using OS Open, call the IBM PowerPC Embedded Systems Solutions Center at (919) 254-1810.

Where to Find More Information

The publications listed under the following headings contain information about the OS Open operating system, or are referred to in this book.

Related IBM Publications

The following titles comprise the OS Open library. To order, contact your IBM Microelectronics representative or call the IBM PowerPC Embedded Systems Solutions Center at (919) 254-1810. Order numbers follow the titles.

OS Open Library

IBM OS Open Programmer's Reference, Volume 1, 92G6911

IBM OS Open Programmer's Reference, Volume 2, 92G6912

IBM OS Open User's Guide, 92G6897, provides detailed information about using the OS Open operating system.

Companion publications to *IBM OS Open User's Guide* describe how to install, configure, and run the OS Open operating system on specific hardware platforms. Each platform-specific book also contains descriptions of platform-specific C functions.

Standards

This book refers to standards and draft standards from the American National Standards Institute (ANSI) and the Institute of Electrical and Electronics Engineers (IEEE).

ANSI C Standard

The following standard can be ordered from ANSI, 1430 Broadway, New York, NY 10018.

American National Standard for Information Systems – Programming Language – C, ANSI X3.159, 1989, American National Standards Institute.

IEEE POSIX Standard and Draft Standards

The following standard and draft standards may be ordered from IEEE, 445 Hoes Lane, Piscataway, NJ 08855-1331.

Information Technology - Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language], IEEE Std. 1003.1b-1993, Institute of Electrical and Electronics Engineers, Inc.

Draft Standard for Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) – Amendment 2: Threads Extension [C Language], P1003.1c Draft 9, May 1994, Institute of Electrical and Electronics Engineers, Inc.

Draft Standard for Information Technology - Standardized Application Environment Profile – POSIX Realtime Application Support (AEP), P1003.13 Draft 6, June 1994, Institute of Electrical and Electronics Engineers, Inc.

Chapter 1. OS Open Function and Macro Summaries

This chapter summarizes OS Open* functions and macros in the following tables, which are organized within the following categories:

- System management
- Interrupt handling
- Signal handling
- Thread management
 - Thread handling
 - Thread attributes handling
 - Thread condition handling
 - Thread group handling
 - Rate monotonic scheduling
- Inter-thread communication
 - Condition handling
 - Message queue handling
 - Mutex handling
 - Semaphore handling
 - Shared memory handling
- Resource management
 - Clock and timers
 - Memory allocation
 - Virtual memory management
 - Memory map, heap, and pool management
 - Devices
 - TTYs
 - Files
 - Input/Output
- TCP/IP networking and networking management
- Remote Procedure Call (RPC) support
- Programming support
 - Loader operations
 - Symbol table operations
 - Character handling utilities
 - Math utilities
 - Queue handling utilities
 - Ring buffer utilities
 - String handling utilities
 - General programming utilities
 - Debugging and reporting

- OpenShell

The page number in each table entry refers to the complete function or macro description in Chapter 2, "OS Open Function and Macro Descriptions."

In the following tables, the column "Appl" indicates whether or not the function may be called from an application thread. If "No", the function should only be used from the kernel thread group.

System Management Functions

Table 1. System Management Functions

Function or Macro	Description	Library or Definition	Appl.	Page
abort()	Raises the SIGABRT signal and then abnormally terminates OS Open	cLib.a	Yes	50
assert()	Writes information to the standard error if an expression is false and calls abort()	<assert.h>	Yes	67
setsysconf()	Alters values returned by sysconf()	rtxLib.a	Yes	728
setuname()	Alters fields in the structure returned by uname()	rtxLib.a	Yes	729
sysconf()	Determines the value of a configurable system limit or option	rtxLib.a	Yes	841
sysconfig()	Installs and removes supervisor service functions	rtxLib.a	No	844
uname()	Stores information identifying the operating system	rtxLib.a	Yes	920

Interrupt Handling Functions

Table 2. Interrupt Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
int_disable()	Disables an event	rtxLib.a	No	320
int_enable()	Enables an event	rtxLib.a	No	321
int_flihdisable()	Calls the user installed disable function	ioLib.a	No	322
int_flihenable()	Calls the user installed enable function	ioLib.a	No	324
int_flihregister()	Registers user supplied functions <i>enablefunc()</i> and <i>disablefunc()</i> to be called when int_flihenable() and int_flihdisable() are called	ioLib.a	No	325

Table 2. Interrupt Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
int_getid()	Returns the current thread's thread ID , or 0 if called from an external interrupt handler Usage note: Called from FLIH	rtxLib.a	N/A	326
int_install()	Installs a FLIH for an event listed in the file <flih.h>	rtxLib.a	No	327
int_query()	Returns information about the FLIH, if any, for an event	rtxLib.a	No	329

Signal Handling Functions

Table 3. Signal Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
kill()	Sends a signal to a process	rtxLib.a	Yes	350
pthread_kill()	Delivers a signal asynchronously to a thread	rtxLib.a	Yes	568
pthread_sigmask()	Examines or changes (or both) the signal mask of the calling thread	rtxLib.a	Yes	594
raise()	Sends a signal to a process	cLib.a	Yes	630
sigaddset()	Adds a signal to a signal set	rtxLib.a	Yes	740
sigdelset()	Deletes a signal from a signal set	rtxLib.a	Yes	741
sigemptyset()	Initializes a signal set to exclude all valid OS Open signals	rtxLib.a	Yes	742
sigfillset()	Initializes a signal set to include all valid OS Open signals	rtxLib.a	Yes	743
sigismember()	Tests whether a signal is a member of a signal set	rtxLib.a	Yes	744
signal()	Handles signals or calls a signal-handling function	cLib.a	Yes	746
sigpending()	Stores the union of the signals blocked for delivery and pending on OS Open and the thread into a signal set object	rtxLib.a	Yes	748
sigwait()	Waits for a signal s in a signal set, clears it from the set of pending signals, and returns its signal number	rtxLib.a	Yes	751

Thread Management Functions

Table 4. Thread Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
atexit()	Registers a function to be called at normal thread termination Usage Note: Unique to thread	cLib.a	Yes	70
exit()	Calls pthread_exit() , which terminates the calling thread	<std.h>	Yes	187
pthread_cancel()	Requests cancelation of a thread	rtxLib.a	Yes	533
pthread_cleanup_pop()	Pops the routine at the top of the calling thread's cleanup stack and conditionally executes the routine	<pthread.h>	Yes	534
pthread_cleanup_push()	Pushes a cleanup routine onto the calling thread's cleanup stack	<pthread.h>	Yes	535
pthread_create()	Creates and initializes a new thread and stores its identifier	rtxLib.a	Yes	552
pthread_detach()	Releases storage allocated for a thread	rtxLib.a	Yes	554
pthread_equal()	Compares two thread IDs	rtxLib.a	Yes	555
pthread_exit()	Terminates the calling thread and makes its return code available to a join with the terminating thread	rtxLib.a	Yes	557
pthread_getschedparam()	Retrieves the scheduling policy and the scheduling parameters of a thread	rtxLib.a	Yes	560
pthread_getspecific()	Gets the value currently bound to a key on behalf of the calling thread	rtxLib.a	Yes	561
pthread_getstackaddr_np()	Retrieves stack bottom of the current thread	rtxLib.o	Yes	562
pthread_join()	Blocks the calling thread until the target thread terminates	rtxLib.a	Yes	563
pthread_key_create()	Creates a key visible to all threads	rtxLib.a	Yes	564
pthread_key_delete()	Deletes a thread-specific data key previously returned by pthread_key_create()	rtxLib.a	Yes	567
pthread_resume_np()	Resumes a thread that was suspended	rtxLib.a	Yes	588

Table 4. Thread Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
pthread_setcancelstate()	Sets the calling thread's cancelability state and returns the previous cancelability state	rtxLib.a	Yes	590
pthread_setcanceltype()	Sets the calling thread's cancelability type and returns the previous cancelability type	rtxLib.a	Yes	591
pthread_setschedparam()	Sets the scheduling policy and associated scheduling parameters for a thread	rtxLib.a	Yes	592
pthread_setspecific()	Associates a thread-specific value with a key obtained by a call to pthread_key_create()	rtxLib.a	Yes	593
pthread_suspend_np()	Suspends a running or ready thread	rtxLib.a	Yes	596
pthread_testcancel()	Creates a cancellation point in the calling thread	rtxLib.a	Yes	597
sched_getmask()	Retrieves the scheduling mask	rtxLib.a	Yes	687
sched_setmask()	Sets the scheduling mask	rtxLib.a	Yes	688
sched_yield()	Causes the calling thread to yield its processor to a ready thread of the same priority	rtxLib.a	Yes	689

Thread Attributes Handling Functions

Table 5. Thread Attributes Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
pthread_attr_getallocstack_np()	Gets the value of the stackalloc attribute from a thread attributes object	rtxLib.a	Yes	507
pthread_attr_getdetachstate()	Gets the value of the detachstate attribute from a thread attributes object	rtxLib.a	Yes	508
pthread_attr_getfp_np()	Stores the floating point availability flag of a thread attributes object in a specific location	rtxLib.a	Yes	509
pthread_attr_getinheritsched()	Gets the value of the inheritsched attribute from a thread attributes object	rtxLib.a	Yes	510

Table 5. Thread Attributes Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
pthread_attr_getpr_np()	Gets the privilege status of the current thread and stores it in a variable	rtxLib.a	Yes	511
pthread_attr_getschedparam()	Gets scheduling parameter attributes from a thread attributes object	rtxLib.a	Yes	512
pthread_attr_getschedpolicy()	Gets the value of the schedpolicy attribute from a thread attributes object	rtxLib.a	Yes	513
pthread_attr_getscope()	Gets the value of the contentionscope attribute from a thread attributes object	rtxLib.a	Yes	514
pthread_attr_getstacksize()	Gets the value of the stacksize attribute from a thread attributes object	rtxLib.a	Yes	516
pthread_attr_getstackaddr()	Gets the value of the stackaddr attribute from a thread attributes object	rtxLib.a	Yes	515
pthread_attr_gettg_np()	Gets the value of the tg attribute from a thread attributes object	rtxLib.a	Yes	517
pthread_attr_gettgid_np()	Gets the value of the tgid attribute from a thread attributes object	rtxLib.a	Yes	518
pthread_attr_init()	Creates and initializes a thread attributes object and stores its identifier	rtxLib.a	Yes	519
pthread_attr_setallocstack_np()	Sets the stackalloc attribute in a thread attributes object	rtxLib.a	Yes	520
pthread_attr_setdetachstate()	Sets the detachstate attribute in a thread attributes object	rtxLib.a	Yes	521
pthread_attr_setfp_np()	Sets the floating point availability flag of a thread attributes object	rtxLib.a	Yes	522
pthread_attr_settg_np()	Sets the tg attribute in a thread attributes object	rtxLib.a	Yes ¹	530
pthread_attr_settgid_np()	Sets the tgid attribute in a thread attributes object	rtxLib.a	Yes	532
pthread_attr_setinheritsched()	Sets the inheritsched attribute in a thread attributes object	rtxLib.a	Yes	523
pthread_attr_setschedparam()	Sets scheduling parameter attributes in a thread attributes object	rtxLib.a	Yes	524

Table 5. Thread Attributes Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
pthread_attr_setschedpolicy()	Sets the schedpolicy attribute in a thread attributes object	rtxLib.a	Yes	526
pthread_attr_setscope()	Sets the contentionscope attribute in a thread attributes object	rtxLib.a	Yes	527
pthread_attr_setstackaddr()	Sets the stackaddr attribute in a thread attributes object	rtxLib.a	Yes	528
pthread_attr_setstacksize()	Sets the stacksize attribute in a thread attributes object	rtxLib.a	Yes	529

1. If the thread group parameter specifies **PTHREAD_SET_TG_NP**, the function cannot be called from an application thread.

Thread Condition Handling Functions

Table 6. Thread Condition Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
pthread_cond_broadcast()	Unblocks all threads blocked on a condition variable	rtxLib.a	Yes	538
pthread_cond_destroy()	Destroys a condition variable	rtxLib.a	Yes	539
pthread_cond_init()	Creates a condition variable and stores it in an object	rtxLib.a	Yes	540
pthread_cond_signal()	Unblocks a thread blocked on a condition variable	rtxLib.a	Yes	543
pthread_cond_timedwait()	Blocks the calling thread, subject to a timeout, on a condition variable	rtxLib.a	Yes	544
pthread_cond_wait()	Blocks the calling thread on a condition variable	rtxLib.a	Yes	546

Thread Condition Attribute Handling Functions

Table 7. Thread Condition Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
pthread_condattr_destroy()	Destroys a condition attributes object	rtxLib.a	Yes	548
pthread_condattr_getpshared()	Gets the pshared attribute from a condition attributes object	rtxLib.a	Yes	549

Table 7. Thread Condition Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
pthread_condattr_init()	Creates and initializes a condition attributes object and stores its identifier	rtxLib.a	Yes	550
pthread_condattr_setpshared()	Sets the pshared attribute in a condition attributes object	rtxLib.a	Yes	551

Thread Group Handling Functions

Table 8. Thread Group Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
istgid()	Returns nonzero (true) if <i>tgid</i> is a valid thread group ID; otherwise, returns 0 (false).	kadtLib.a	Yes	346
pthread_tgcreate_np()	Creates a new thread group	rtx.o	Yes	598
pthread_tgdestroy_np()	Destroys a thread group	rtx.o	Yes	599
pthread_tggetkernel_np()	Returns the thread group ID of the PTHREAD_KERNEL_NP thread group	rtx.o	Yes	600
pthread_tggetspecific_np()	Gets the value currently bound to a specified key and the current thread group	rtx.o	Yes	601
pthread_tgkeycreate_np()	Creates a key, visible to all thread groups, and stores the key value in an object	rtx.o	No	603
pthread_tgkeydelete_np()	Deletes a thread group-specific data key	rtx.o	No	605
pthread_tgself_np()	Returns the thread group ID of the calling thread	rtx.o	Yes	606
pthread_tgsetspecific_np()	Associates a thread group-specific <i>value</i> with the current thread group and a key	rtx.o	Yes	607
pthread_whattg_np()	Returns the thread group ID of a thread	rtx.o	Yes	608
tg_dump()	Provides information about a thread group	kadtLib.a	No	874
tg_dump_all()	Provides information about all thread groups.	kadtLib.a	No	876
tg_list()	Lists all thread groups and identifies the kernel thread group	kadtLib.a	No	878

Rate Monotonic Scheduler Functions

Table 9. Rate Monotonic Scheduler Functions

Function or Macro	Description	Library or Definition	Appl	Page
rms_delq()	Removes the oldest node in the delta queue used by the rate monotonic scheduler (RMS) thread Usage Note: Wrapper may be started as an Application group	rmsLib.a	No	655
rms_init()	Initializes the delta queue used by the RMS thread	rmsLib.a	No	656
rms_insq()	Inserts a node into the delta queue used by the RMS thread	rmsLib.a	No	657
rms_oldest()	Returns the address of the oldest node in the delta queue used by the RMS thread	rmsLib.a	No	658
rms_start()	Starts the RMS thread, which manages all user-supplied periodic threads	rmsLib.a	No	659

Message Queue Handling Functions

Table 10. Message Queue Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
mq_close()	Removes the association between a message queue descriptor and a message queue	rtxLib.a	Yes	435
mq_getattr()	Retrieves a message queue attribute from a message queue	rtxLib.a	Yes	437
mq_notify()	Associates a notification request with a message queue descriptor	rtxLib.a	Yes	439
mq_open()	Creates a message queue descriptor that establishes a connection between the system process and a message queue	rtxLib.a	Yes	441
mq_receive()	Receives the oldest of the highest priority messages from a message queue	rtxLib.a	Yes	446
mq_send()	Adds a message to a message queue	rtxLib.a	Yes	449
mq_setattr()	Sets the attributes associated with a message queue	rtxLib.a	Yes	452
mq_timedreceive()	Receives the oldest of the highest priority messages within a time interval or times out	rtxLib.a	Yes	453

Table 10. Message Queue Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
mq_timedsend()	Sends the oldest of the highest priority messages within a time interval or times out	rtxLib.a	Yes	455
mq_unlink()	Removes a message queue	rtxLib.a	Yes	457

Mutex Handling Functions

Table 11. Mutex Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
pthread_mutex_destroy()	Destroys a mutex	rtxLib.a	Yes	569
pthread_mutex_getprioceiling()	Gets the prioceiling attribute for a mutex	rtxLib.a	Yes	570
pthread_mutex_init()	Creates and initializes a mutex, and stores its identifier in an object	rtxLib.a	Yes	578
pthread_mutex_lock()	Locks a mutex	rtxLib.a	Yes	572
pthread_mutex_setprioceiling()	Changes the priority ceiling attribute of a mutex	rtxLib.a	Yes	573
pthread_mutex_timedlock()	Locks a mutex or times out if unable to acquire the mutex to be locked	rtxLib.a	Yes	574
pthread_mutex_trylock()	Conditionally locks a mutex	rtxLib.a	Yes	575
pthread_mutex_unlock()	Unlocks a mutex	rtxLib.a	Yes	576

Mutex Attributes Handling Functions

Table 12. Mutex Attributes Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
pthread_mutexattr_destroy()	Destroys a mutex attributes object	rtxLib.a	Yes	577
pthread_mutexattr_getprioceiling()	Gets the prioceiling attribute from a mutex attributes object	rtxLib.a	Yes	578
pthread_mutexattr_getprotocol()	Gets the protocol attribute from a mutex attributes object	rtxLib.a	Yes	579
pthread_mutexattr_getpshared()	Gets the pshared attribute from a mutex attributes object	rtxLib.a	Yes	580

Table 12. Mutex Attributes Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
pthread_mutexattr_init()	Creates and initializes a mutex attributes object	rtxLib.a	Yes	581
pthread_mutexattr_setprioceiling()	Sets the prioceiling attribute in a mutex attributes object	rtxLib.a	Yes	584
pthread_mutexattr_setpshared()	Gets the pshared attribute from the mutex attributes object	rtxLib.a	Yes	586
pthread_mutexattr_setprotocol()	Sets the protocol attribute in a mutex attributes object	rtxLib.a	Yes	579

Semaphore Handling Functions

Table 13. Semaphore Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
sem_close()	Indicates that OS Open is finished using a named semaphore	rtxLib.a	Yes	696
sem_destroy()	Destroys an unnamed semaphore descriptor	rtxLib.a	Yes	697
sem_getvalue()	Retrieves the value of a semaphore	rtxLib.a	Yes	698
sem_init()	Initializes an unnamed semaphore descriptor	rtxLib.a	Yes	699
sem_open()	Establishes a connection between a named semaphore and the system process	rtxLib.a	Yes	700
sem_post()	Increments a counting semaphore	rtxLib.a	Yes	705
sem_timedwait()	Decrements a counting semaphore or times out if blocked on the semaphore	rtxLib.a	Yes	706
sem_trywait()	Conditionally decrements a counting semaphore	rtxLib.a	Yes	707
sem_unlink()	Marks a named semaphore as "destructible"	rtxLib.a	Yes	708
sem_wait()	Decrements a counting semaphore	rtxLib.a	Yes	709

Shared Memory Handling Functions

Table 14. Shared Memory Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
mmap()	Establishes a mapping between the OS Open address space and a memory object	devLib.a	Yes	396
munmap()	Removes any mappings for pages in an address range	devLib.a	Yes	461
shm_open()	Establishes a connection between a shared memory object and a file descriptor	devLib.a	Yes	734
shm_unlink()	Removes the name of a shared memory object	devLib.a	Yes	738

Clock and Timer Functions

Table 15. Clock and Timer Functions

Function or Macro	Description	Library or Definition	Appl	Page
alarm()	Instructs OS Open to deliver SIGALRM to the system (OS Open) process after a period elapses	rtxLib.a	Yes	58
asctime()	Converts time to a character string Usage Note: Use <code>asctime_r()</code>	cLib.a	No	62
asctime_r()	Reentrant version of asctime()	cLib.a	Yes	64
clock()	Returns -1; provided for ANSI C compatibility	cLib.a	Yes	123
clock_getres()	Gets the resolution (in nanoseconds) of a clock	rtxLib.a	Yes	124
clock_gettime()	Gets the current time from a clock and stores the time	rtxLib.a	Yes	125
clock_settime()	Sets a clock	rtxLib.a	Yes	126
ctime()	Converts a time value to local time in a character string Usage Note: Use <code>ctime_r()</code>	cLib.a	No	149
ctime_r()	Reentrant version of ctime()	cLib.a	Yes	150
difftime()	Computes the difference between two calendar times	cLib.a	Yes	175
gmtime()	Converts a time value to a structure Usage Note: Use <code>gmtime_r()</code>	cLib.a	No	299
gmtime_r()	Reentrant version of gmtime()	cLib.a	Yes	301

Table 15. Clock and Timer Functions

Function or Macro	Description	Library or Definition	Appl	Page
localtime()	Converts calendar time to broken-down time, expressed as local time Usage Note: Use localtime_r()	cLib.a	No	388
localtime_r()	A reentrant version of localtime()	cLib.a	Yes	389
mktime()	Converts local time, stored as a structure, into a time_t type suitable for use with other time functions	cLib.a	Yes	429
nanosleep()	Blocks a thread for a number of nanoseconds	rtxLib.a	Yes	466
sleep()	Suspends a thread for an interval, or until an unblocked signal is delivered to the thread	rtxLib.a	Yes	755
strftime()	Places characters into an array according to a format string	cLib.a	Yes	795
time()	Returns calendar time as the number of seconds since January 1, 00:00:00 universal coordinated time (UTC)	cLib.a	Yes	886
timer_create()	Creates a timer using CLOCK_REALTIME	rtxLib.a	Yes	888
timer_delete()	Deletes a timer	rtxLib.a	Yes	890
timer_dump()	Provides information about a timer	kadtLib.a	Yes	891
timer_getoverrun()	Returns the timer overrun count for a timer	rtxLib.a	Yes	892
timer_gettime()	Returns time until expiration and reload value for a timer	rtxLib.a	Yes	893
timer_list()	Provides information about all timers	kadtLib.a	Yes	895
timer_settime()	Arms or disarms a timer	rtxLib.a	Yes	897
timertick_notify()	Notifies OS Open that a timer tick expired	rtxLib.a	Yes	899

Memory Allocation Functions

Table 16. Memory Allocation Functions

Function or Macro	Description	Library or Definition	Appl	Page
calloc()	Allocates memory for an array and initializes each element to 0 Usage Note: cLib_init() must be run run first	cLib.a	Yes	89
free()	Frees a block of storage Usage Note: cLib_init() must be run run first	cLib.a	Yes	228

Table 16. Memory Allocation Functions

Function or Macro	Description	Library or Definition	Appl	Page
malloc()	Allocates memory Usage Note: <code>cLib_init()</code> must be run first	cLib.a	Yes	398
realloc()	Changes the size of an allocated block of memory Usage Note: <code>cLib_init()</code> must be run first	cLib.a	Yes	638

Virtual Memory Management Functions

Table 17. Virtual Memory Management Functions

Function or Macro	Description		Appl	Page
vm_addrealpages()	Allocates real memory in a virtual memory space	rtx.o	Yes ¹	932
vm_allocate()	Allocates virtual memory space for a thread group	rtx.o	Yes ¹	935
vm_copy()	Copies bytes from a source address to a destination address in a thread group	rtx.o	Yes ¹	937
vm_deallocate()	Deallocates virtual memory space for a thread group	rtx.o	Yes ¹	939
vm_protect()	Sets the protection attributes for a virtual memory range	rtx.o	Yes ¹	941
vm_query()	Returns Translation Manager statistics to a calling application	rtx.o	Yes	943
vm_read()	Transfers bytes from a thread group to the current thread group	rtx.o	No	945
vm_region()	Searches for the next virtual memory region	rtx.o	Yes	947
vm_removealpages()	Removes mappings for real memory in a virtual memory space	rtx.o	Yes ¹	949
vm_translate()	Returns a real memory address corresponding to a virtual address in a thread group	rtx.o	Yes ¹	951
vm_write()	Transfers bytes from the current thread group to an address in a specified thread group	rtx.o	No	953

1. If the thread group parameter specifies **PTHREAD_KERNEL_NP**, the function cannot be called from an application thread.

Memory Map Management Functions

Table 18. Memory Map Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
mlock()	Causes all pages mapped by OS Open in a range to be memory resident until unlocked	devLib.a	Yes	459
mlockall()	Causes all pages mapped by OS Open to be memory resident until unlocked	devLib.a	Yes	431
munlock()	Unlocks pages mapped by OS Open starting at an address	devLib.a	Yes	459
munlockall()	Unlocks all pages mapped by OS Open	devLib.a	Yes	460

Memory Heap Management Functions

Table 19. Memory Heap Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
memheap_alloc()	Allocates memory from a heap	rtxLib.a	Yes	405
memheap_alloc_aligned()	Allocates boundary-aligned memory from a heap	rtxLib.a	Yes	406
memheap_alloc_pages()	Allocates page-aligned memory from a heap	rtxLib.a	Yes	407
memheap_extend()	Adds a new segment to a heap and makes its free space available for allocation	rtxLib.a	Yes	408
memheap_free()	Frees previously allocated memory	rtxLib.a	Yes	409
memheap_query()	Returns the status of a heap	rtxLib.a	Yes	410
memheap_realloc()	Changes the size of an allocated block of memory	rtxLib.a	Yes	413
memheap_replace()	Replaces OS Open memory management functions with user-supplied functions Usage Note: Replaced functions are global to all thread groups	rtxLib.a	Yes	414

Memory Pool Management Functions

Table 20. Memory Pool Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
mempool_alloc()	Allocates memory from a buffer pool	rtxLib.a	Yes	417

Table 20. Memory Pool Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
mempool_destroy()	Removes a buffer pool from use	rtxLib.a	Yes	418
mempool_free()	Returns memory to a buffer pool	rtxLib.a	Yes	419
mempool_init()	Creates a buffer pool	rtxLib.a	Yes	420
mempool_query()	Returns pool statistics into a structure	rtxLib.a	Yes	422

Device Management Functions

Table 21. Device Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
dev_io_init()	Prepares a device subsystem for use and initializes the NULL and shared memory device drivers	rtxLib.a	No	171
device_install()	Associates a device driver with a device name in the root directory managed by the device subsystem	rtxLib.a	No	172
device_uninstall	Removes a device that was installed with device_install()	devLib.a	No	174
driver_install()	Identifies a device driver to the device subsystem	rtxLib.a	No	177
ioctl()	Uses an open file descriptor to issue device-specific control commands	devLib.a	Yes	332
select()	Allows an application to query a file (usually a special file) to determine whether there is data to read, space to write, or outstanding exception conditions	devLib.a	Yes	691
select_notify()	Allows a device driver to signal a previously requested notification of a select() status change	devLib.a	No	694
select_redrive()	Used internally by device drivers to redrive a device-specific select() to another device	devLib.a	No	695
strategy()	The interface for application users of block special files to request blocks of data to be written or read	devLib.a	Yes	783

PCMCIA Handling Functions

Table 22. PCMCIA Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
cs_card_status()	Returns information about the insertion state of a PC Card and stores the insertion state information	csLib.a	Yes	138
cs_get_io_window()	Reserves an available I/O window for a PCMCIA socket	csLib.a	No	139
cs_get_mem_window()	Reserves an available memory window for a PCMCIA socket	csLib.a	No	140
cs_init()	Initializes the PCMCIA Card Services/Enabler (CS/Enabler) software layer	csLib.a	No	141
cs_int_install()	Installs a first-level interrupt handler (FLIH) when multiple cards share an OS Open event	csLib.a	No	143
cs_int_query()	Returns information about a FLIH when multiple cards share an OS Open event	csLib.a	No	144
cs_pata_parm_init()	Initializes configurable parameters for a PCMCIA ATA/IDE (PATA) device	csLib.a	No	145
cs_stateless_card_status()	Returns information about the insertion state of a PC Card	csLib.a	No	148
pata_init()	Initializes PATA device driver support for IBM and Integral 105 MB and 260 MB hard disk drives	csLib.a	No	485
socket_services()	Invokes Socket Services selected by supplied parameters	csLib.a	No	762
ss_init()	Initializes Socket Services	csLib.a	No	766

TTY Handling Functions

Table 23. TTY Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
cfgetispeed()	Returns a terminal input baud rate	ttylib.a	Yes	92
cfgetospeed()	Returns a terminal output baud rate	ttylib.a	Yes	93
cfsetispeed()	Sets a terminal input baud rate	ttylib.a	Yes	94
cfsetospeed()	Sets a terminal output baud rate	ttylib.a	Yes	95

Table 23. TTY Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
clearDisp()	Clears the terminal screen (VT100 terminals and emulations only)	shell.o	Yes	98
isatty()	Returns 1 if the argument is a valid file descriptor associated with a terminal	ttylib.a	Yes	336
tcdrain()	Waits until all output written to a tty object is transmitted	ttylib.a	Yes	849
tcflow()	Suspends data reception on a tty object	ttylib.a	Yes	850
tcflush()	Discards any data written to a tty object but not transmitted, or data received but not read	ttylib.a	Yes	852
tcgetattr()	Gets parameters of an object and stores them in the structure termios	ttylib.a	Yes	854
tcgetkey()	Copies, to a buffer, the string assigned to a function key	ttylib.a	Yes	856
tcsendbreak()	Sends a continuous stream of zero-valued bits for a specified period	ttylib.a	Yes	858
tcsetattr()	Sets parameters associated with a terminal from the structure termios	ttylib.a	Yes	860
tcsetkey()	Associates a character string with a function key	ttylib.a	No	862
tty_init()	Initializes the general terminal interface for use	ttylib.a	Yes	915

File Handling Functions

Table 24. File Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
access()	Checks a file's access permissions	devLib.a	Yes	55
chmod()	Changes the file permission bits of a file	devLib.a	Yes	97
clearerr()	Resets error and end-of-file (EOF) indicators for a stream	fsLib.a	Yes	99
close()	Deallocates a file descriptor	devLib.a	Yes	126
closedir()	Closes a directory stream	devLib.a	Yes	128
fclose()	Closes a stream and flushes all buffers associated with the stream before closing it	fsLib.a	Yes	190
fcntl()	Provides control commands for open files	devLib.a	Yes	191

Table 24. File Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
fdopen()	Opens a stream and associates it with a file descriptor	fsLib.a	Yes	193
feof()	Tells whether the EOF flag is set for a stream	fsLib.a	Yes	195
ferror()	Tests for an error in reading from or writing to a stream	fsLib.a	Yes	196
fflush()	Empties the buffer associated with an output stream	fsLib.a	Yes	197
fgetpos()	Stores the value of the file position indicator of a stream in an object	fsLib.a	Yes	199
fileno()	Returns the file descriptor associated with a stream	<stdio.h>	Yes	202
flockfile()	Called by a thread to lock (get exclusive access to) a stream	fsLib.a	Yes	205
fopen	Opens a named file and associates a stream with it	fsLib.a	Yes	210
format()	Places an empty FAT file system on a block device	fatLib.a	Yes	215
fpathconf()	Retrieves the value of a configurable limit or option associated with an open file	devLib.a	Yes	217
freopen()	Closes the file associated with a stream and reassigns the stream to another file	fsLib.a	Yes	229
fs_init()	Initializes the C Library file system	fsLib.a	No	231
fseek()	Sets the file position indicator of the input stream to a location offset bytes from its origin	fsLib.a	Yes	235
fsetpos()	Sets the file position indicator of a stream to a location obtained by a previous call to fgetpos()	fsLib.a	Yes	237
fstat()	Stores information about an open file in an object	devLib.a	Yes	238
ftell()	Returns the value of the file position indicator associated with a stream	fsLib.a	Yes	239
ftruncate()	Sets the size of a file	devLib.a	Yes	250
ftrylockfile()	Non-blocking version of flockfile()	fsLib.a	Yes	251
funlockfile()	Relinquishes the lock (exclusive access) to a stream	fsLib.a	Yes	252
link()	Creates a directory entry and associates it with a file	devLib.a	Yes	384

Table 24. File Handling Functions

Function or Macro	Description	Library or Definition	Appl	Page
lseek()	Sets the file offset for an open file descriptor	devLib.a	Yes	393
mkdir()	Creates an empty directory	devLib.a	Yes	427
open()	Establishes a connection between a file and a file descriptor	devLib.a	Yes	478
opendir()	Opens a directory stream corresponding to a directory name	devLib.a	Yes	480
pathconf()	Retrieves the value of a configurable limit or option of a file	devLib.a	Yes	486
readdir()	Returns a pointer to a structure representing a directory entry Usage Note: Use readdir_t()	devLib.a	No	636
readdir_r()	Stores a pointer to a structure representing a directory entry	devLib.a	Yes	637
rename()	Changes the name of a file or directory	devLib.a	Yes	650
rewind()	Repositions the file position indicator associated with a stream at the beginning of a file and clears the error indicator for the stream	fsLib.a	Yes	652
rewinddir()	Resets the position indicator of a directory stream to the first entry	devLib.a	Yes	653
rmdir()	Removes a directory	devLib.a	Yes	654
setbuf()	Controls buffering for a stream	fsLib.a	Yes	720
setvbuf()	Controls buffering and buffer size for a stream	fsLib.a	Yes	730
stat()	Stores information about a file in an object	devLib.a	Yes	770
tmpfile()	Creates a temporary binary file that is removed when closed	fsLib.a	Yes	900
tmpnam()	Creates a unique valid file name	fsLib.a	Yes	901
unlink()	Removes a link to a file and decrements the file's link count	devLib.a	Yes	923
utime()	Sets the access and modification times of a file	devLib.a	Yes	924

Root Directory Entry Management Functions (Virtual Memory Only)

Table 25. Root Directory Entry Management Functions (Virtual Memory Only)

Function or Macro	Description	Library or Definition	Appl	Page
rootname_add()	Adds a root directory entry to the root directory	rtx.o	Yes	670
rootname_delete()	Removes a root directory entry	rtx.o	Yes	671
rootname_get()	Returns user-specific information about a root directory entry	rtx.o	Yes	672
rootname_query()	Retrieves consecutive root directory entries	rtx.o	Yes	673

Input/Output Functions

Table 26. Input/Output Functions

Function or Macro	Description	Library or Definition	Appl	Page
bprintf()	Prints a formatted string to a buffer, or calls a print function previously set by bprintf_set()	kadtLib.a	Yes	84
bprintf_set()	Sets the print function for bprintf()	kadtLib.a	No	85
fgetc()	Returns the next character from an input stream and advances the file position indicator	fsLib.a	Yes	198
fgets()	Reads a string from an input stream and stores it in a buffer	fsLib.a	Yes	201
fprintf()	Formats and writes a series of characters and values to an output stream	fsLib.a	Yes	221
fputc()	Converts an integer to an unsigned character, writes the character to a stream, and advances the file position indicator	fsLib.a	Yes	222
fputs()	Writes a string into a stream	fsLib.a	Yes	224
fread()	Reads items from an input stream, stores them in a buffer, and advances the file position indicator by the number of bytes read	fsLib.a	Yes	225
fscanf()	Reads data from a stream into locations given by entries in the argument list, if any	fsLib.a	Yes	234
fwrite()	Writes items from a buffer to an output stream	fsLib.a	Yes	253
getc()	Returns the next character from a input stream and advances the file position indicator	<stdio.h>	Yes	255
getc_unlocked()	Equivalent to getc() , except that getc_unlocked() is not async safe	fsLib.a	Yes	256

Table 26. Input/Output Functions

Function or Macro	Description	Library or Definition	Appl	Page
getchar()	Reads a character from the standard input at the file position indicator, returns the character read, and advances the file position indicator to the next character	<stdio.h>	Yes	257
getchar_unlocked()	Equivalent to getchar() , except that getchar_unlocked() is not async safe	<stdio.h>	Yes	258
gets()	Reads a line from the standard input and stores it in a buffer	fsLib.a	Yes	289
printf()	Formats and prints a series of characters and values to the standard output	fsLib.a	Yes	500
putc()	Converts an integer to an unsigned char and writes the character to the current position of an output stream	<stdio.h>	Yes	609
putc_unlocked()	Equivalent to putc() , except that putc_unlocked() is not async safe	fsLib.a	Yes	611
putchar()	Converts an integer to an unsigned char and writes the character to the standard output	<stdio.h>	Yes	612
putchar_unlocked()	Equivalent to putchar() , except that it is not async safe	<stdio.h>	Yes	613
puts()	Writes a string to the standard output and appends a new line to the output	fsLib.a	Yes	615
read()	Tries to read bytes from a file, associated with an open file descriptor, into a buffer	devLib.a	Yes	633
scanf()	Reads data from the standard input into locations given by each entry in an argument list	fsLib.a	Yes	681
sprintf()	Formats and stores a series of characters and values in a buffer	cLib.a	Yes	763
sscanf()	Reads data from a buffer into the locations given by the argument list	cLib.a	Yes	768
ungetc()	Pushes a character onto the given input stream	fsLib.a	Yes	921
vfprintf()	Formats and writes a series of characters and values, specified by a variable argument list, to a stream	fsLib.a	Yes	930
vprintf()	Formats and prints a series of characters and values, specified by a variable argument list, to the standard output	fsLib.a	Yes	955

Table 26. Input/Output Functions

Function or Macro	Description	Library or Definition	Appl	Page
vsprintf()	Formats and stores a series of characters and values, specified by a variable argument list, in a buffer	cLib.a	Yes	956
write()	Writes from a buffer to a file associated with a descriptor	devLib.a	Yes	960

TCP/IP Networking and Network Management Functions

Table 27. TCP/IP Networking and Network Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
accept()	Accepts a connection on a socket and returns a new socket for the connection	tcpipLib.a	Yes	52
bind()	Assigns a name to a unnamed socket	tcpipLib.a	Yes	80
connect()	Connects two sockets	tcpipLib.a	Yes	132
dtom()	Converts the address of the data portion of a memory buffer (mbuf) to the address of the first byte of the mbuf	<sys/mbuf.h>	Yes	178
enet_arpinput()	Provides an Ethernet address resolution protocol (ARP) input routine	tcpipLib.a	No	179
enet_arpresolve()	Resolves an Internet Protocol (IP) address to an Ethernet address	tcpipLib.a	No	181
enet_attach()	Connects an Ethernet interface to TCP/IP protocol stacks	netLib.a	No	183
ftp()	Implements the TCP/IP File Transfer Protocol (FTP) client for file transfers between the local system and a remote system	ftpLib.a	Yes	241
ftpd_start()	Starts the ftpd daemon	ftpLib.a	No	249
gethostbyaddr()	Retrieves an entry from the _Tcpip_hosts database, using a host address as a search key	netLib.a	Yes	263
gethostbyname()	Retrieves an entry from the _Tcpip_hosts database, using a host name as a search key	netLib.a	Yes	265
gethostlock()	Locks (acquires exclusive use of) the _Tcpip_hosts database	netLib.a	Yes	266
gethostname()	Returns the standard host name of a local host	netLib.a	Yes	267

Table 27. TCP/IP Networking and Network Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
gethostunlock()	Unlocks (releases exclusive use of) the _Tcpi p_hosts database	netLib.a	Yes	268
getnetbyaddr()	Retrieves an entry from the _Tcpi p_networks database, using a network address as a search key	netLib.a	Yes	270
getnetbyname()	Retrieves an entry from the _Tcpi p_networks database, using a network name as a search key	netLib.a	Yes	272
getnetlock()	Locks the _Tcpi p_networks database	netLib.a	Yes	273
getnetunlock()	Unlocks the _Tcpi p_networks database	netLib.a	Yes	274
getpeername()	Returns the name of the peer connected to a socket	tcpipLib.a	Yes	277
getprotobyname()	Retrieves an entry from the _Tcpi p_protocols database, using a protocol name as a search key	netLib.a	Yes	280
getprotobynumber()	Retrieves an entry from the _Tcpi p_protocols database, using a protocol number as a search key	netLib.a	Yes	281
getprotolock()	Locks the _Tcpi p_protocols database	netLib.a	Yes	282
getprotounlock()	Unlocks the _Tcpi p_protocols database	netLib.a	Yes	283
getservbyname()	Retrieves an entry from the _Tcpi p_services database, using a service name as a search key	netLib.a	Yes	290
getservbyport()	Retrieves an entry from the _Tcpi p_services database, using a port number as a search key	netLib.a	Yes	291
getservlock()	Locks the _Tcpi p_services database	netLib.a	Yes	293
getservunlock()	Unlocks the _Tcpi p_services database	netLib.a	Yes	294
getsockname()	Returns the name of a socket	tcpipLib.a	Yes	295
getsockopt()	Returns options associated with a socket	tcpipLib.a	Yes	296
if_attach()	Attaches an interface to the list of active interfaces	tcpipLib.a	No	303
ifconfig()	Configures or displays the network interface parameters of a network	netLib.a	No	304
inet_addr()	Translates a string, containing an Internet address in dotted decimal notation, into a 32-bit Internet Protocol (IP) address	netLib.a	Yes	308

Table 27. TCP/IP Networking and Network Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
inet_aton()	Translates a string, containing an Internet address in dotted decimal notation, into a 32-bit IP address	netLib.a	Yes	309
inet_lnaof()	Decomposes an IP address and returns its local portion	netLib.a	Yes	311
inet_makeaddr()	Combines an IP network number and a local network address and returns a new IP address	netLib.a	Yes	312
inet_netof()	Decomposes an IP address and returns its network portion	netLib.a	Yes	313
inet_network()	Translates a string, containing an Internet address in dotted decimal notation, into a 32-bit IP address	netLib.a	Yes	314
inet_ntoa()	Translates an IP address into an ASCII string representing the Internet address in dotted decimal notation Usage Note: Use inet_ntoa_r()	netLib.a	No	316
inet_ntoa_r()	Reentrant version of inet_ntoa()	netLib.a	Yes	317
info_rtable()	Retrieves routing information from the TCP/IP library routing table	tcpipLib.a	No	318
listen()	Listens on a socket for connections and limits the backlog of incoming connection	tcpipLib.a	Yes	386
m_free()	Returns an mbuf to the mbuf pool	tcpipLib.a	No	394
m_freem()	Returns an mbuf chain to the free pool	tcpipLib.a	No	395
M_PREPEND()	Prepends storage to the data portion of an mbuf (macro)	<sys/mbuf.h>	No	396
m_prepend()	Prepends storage to the data portion of an mbuf (function)	tcpipLib.a	No	397
MFREE()	Returns an mbuf to the mbuf pool and retrieves the address of the following mbuf	<sys/mbuf.h>	No	424
MGET()	Allocates an mbuf	<sys/mbuf.h>	No	425
MGETHDR()	Allocates the first (header) mbuf of an mbuf chain	<sys/mbuf.h>	No	426
mtod()	Converts the type of an mbuf pointer	<sys/mbuf.h>	No	458
net_init()	Initializes the netLib.a library	netLib.a	No	468
net_splimp()	Returns a pointer used for serialization within a protocol stack	tcpipLib.a	No	469
net_splx()	Releases a lock placed by net_splimp()	tcpipLib.a	No	470

Table 27. TCP/IP Networking and Network Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
ping()	Sends a message to a network host to obtain a response from a network host or gateway	netLib.a	Yes	489
recv()	Allows an application program to receive messages through a connected socket	netLib.a	Yes	640
recvfrom()	Receives a message from a socket, and can receive data on a socket, whether it is connected or not	tcpipLib.a	Yes	643
recvmsg()	Receives a message from a socket, and can receive data on a socket, whether it is connected or not	tcpipLib.a	Yes	645
route()	Allows for manual manipulations of routing tables	netLib.a	No	674
schednetisr()	Schedules a software network interrupt that signals the arrival of new IP packets	<net/netisr.h>	No	690
send()	Allows an application program to send messages using a connected socket	netLib.a	Yes	713
sendmsg()	Sends messages on connected or unconnected sockets using the msghdr message structure	tcpipLib.a	Yes	716
sendto()	Allows an application program to send messages through an unconnected socket	tcpipLib.a	Yes	718
setsockopt()	Sets options associated with a socket	tcpipLib.a	Yes	725
shutdown()	Shuts down all or part of a full-duplex connection on a socket	tcpipLib.a	Yes	739
slip_attach()	Connects a serial interface to TCP/IP protocol stack	netLib.a	No	756
slip_resume()	Resumes execution of a SLIP receive thread	netLib.a	No	758
slip_suspend()	Suspends execution of a SLIP receive thread	netLib.a	No	759
socket()	Creates an end point for communication and returns a descriptor	tcpipLib.a	Yes	760
tcpip_init()	Initializes the TCP/IP library	tcpipLib.a	No	857
telnetd_assignfunc()	Allows for specification of function other than shell() to be started when user connects to OS Open system	tnetdLib.a	No	863
telnetd_assigntty()	Allows for changing the default tty device name used by the telnet daemon	tnetdLib.a	No	865
telnetd_start()	Starts the telnet server daemon	tnetdLib.a	No	866

Table 27. TCP/IP Networking and Network Management Functions

Function or Macro	Description	Library or Definition	Appl	Page
tftp()	Transfers files between a local host and a remote host Usage Note: <code>tftp_init()</code> must be run first	tftp.o	Yes	870
tn()	Connects the local host with a remote host using TELNET	telnet.o	Yes	902
tok_arpinut()	Provides a token-ring ARP input routine	tcpipLib.a	No	904
tok_arpresolve()	Resolves an IP address into a token-ring address	tcpipLib.a	No	905

Network File System (NFS) Functions

Table 28. Network File System (NFS) Functions

Function or Macro	Description	Library or Definition	Appl	Page
nfs_authget()	Retrieves UNIX authentication parameters	nfsLib.h	Yes	471
nfs_authset()	Sets UNIX authentication parameters	nfsLib.h	No	472
nfs_dinit()	Initializes NFS	nfsLib.h	No	473
nfs_mount()	Creates a local device for a remote file system	nfsLib.h	No	474
nfs_showmount()	Displays exported file systems on a remote host	nfsLib.h	Yes	475
nfs_umount()	Unmounts a local device if no files on the file system are open	nfsLib.h	No	476

Remote Procedure Call Support Functions

Table 29. Remote Procedure Call Support Functions

Function or Macro	Description	Library or Definition	Appl	Page
auth_destroy(), AUTH_DESTROY()	Destroys authentication information	<rpc/auth.h>	Yes	76
authnone_create()	Creates and returns a default RPC authentication handle passing NULL authentication information	rpcLib.a	Yes	77
authunix_create()	Creates an RPC authentication handle having UNIX** permissions	rpcLib.a	Yes	78

Table 29. Remote Procedure Call Support Functions

Function or Macro	Description	Library or Definition	Appl	Page
authunix_create_default()	Calls authunix_create() to create a default UNIX RPC authentication handle	rpcLib.a	Yes	79
callrpc()	Calls a remote procedure on a host	rpcLib.a	Yes	90
clnt_broadcast()	Broadcasts a remote procedure call to all hosts on a network	rpcLib.a	Yes	101
clnt_call(), CLNT_CALL()	Calls the remote procedure associated with a client handle	<rpc/clnt.h>	Yes	103
clnt_control(), CLNT_CONTROL()	Changes or retrieves information about a client object	<rpc/clnt.h>	Yes	104
clnt_create()	Creates and returns a generic client handle	rpcLib.a	Yes	106
clnt_destroy(), CLNT_DESTROY()	Destroys a client handle	<rpc/clnt.h>	Yes	107
clnt_freeres(), CLNT_FREERES()	Frees data previously allocated by RPC/XDR	<rpc/clnt.h>	Yes	108
clnt_geterr(), CLNT_GETERR()	Copies error information from a client RPC handle to an error structure	<rpc/clnt.h>	Yes	109
clnt_pcreateerror()	Writes a message, to the standard error, describing why a client RPC handle was not created	rpcLib.a	Yes	110
clnt_perrno()	Writes a message, to the standard error, specifying the client error status of the remote procedure call	rpcLib.a	Yes	111
clnt_perror()	Writes a message, to the standard error, indicating why a remote procedure call failed	rpcLib.a	Yes	112
clnt_spccreateerror()	Writes an message, to the standard error, describing why a client RPC handle was not created	rpcLib.a	Yes	113
clnt_sperno()	Returns a pointer to a string containing an error message	rpcLib.a	Yes	114
clnt_sperror()	Writes a string to the standard error indicating why a remote procedure call failed	rpcLib.a	Yes	115
clntraw_create()	Creates an RPC client used for simulation	rpcLib.a	Yes	116
clnttcp_create()	Creates an RPC client transport handle, using TCP/IP as the transport, for a remote program	rpcLib.a	Yes	117
clntudp_bufcreate()	Creates an RPC client handle, using UDP as the transport and supporting user-specified buffer sizes, for a remote program	rpcLib.a	Yes	119

Table 29. Remote Procedure Call Support Functions

Function or Macro	Description	Library or Definition	Appl	Page
clntudp_create()	Creates an RPC client transport handle, using UDP as the transport, for a remote program	rpcLib.a	Yes	121
get_myaddress()	Gets a host IP address without accessing the _Tcpi_p_hosts array	rpcLib.a	Yes	254
getrpcbyname()	Retrieves an entry from the _Rpc_rpc database by RPC name	rpcLib.a	Yes	285
getrpcbynumber()	Retrieves an entry from the _Rpc_rpc database by RPC number	rpcLib.a	Yes	286
getrpcent()	Retrieves an entry from the _Rpc_rpc database by entry number	rpcLib.a	Yes	287
getrpcport()	Returns the port number of a remote procedure	rpcLib.a	Yes	288
pmap_getmaps()	Return a list of current RPC port mappings	rpcLib.a	Yes	491
pmap_getport()	Requests the port number on which a specified service waits	rpcLib.a	Yes	492
pmap_rmtcall()	Calls a remote portmap daemon, which makes a remote procedure call	rpcLib.a	Yes	493
pmap_set()	Maps a remote procedure call to a port	rpcLib.a	Yes	495
pmap_unset()	Destroys the mapping between a remote procedure call and a port	rpcLib.a	Yes	496
portmap_thread()	Starts the portmap daemon	rpcLib.a	No	498
registerrpc()	Registers a procedure with the portmap daemon	rpcLib.a	Yes	649
rpc_thread_init()	Initializes thread private storage for use by RPC library functions	rpcLib.a	Yes	676
svc_destroy(), SVC_DESTROY()	Destroys an RPC service transport handle	<rpc/svc.h>	Yes	818
svc_freeargs(), SVC_FREEARGS()	Frees data allocated by RPC/XDR	<rpc/svc.h>	Yes	819
svc_getargs(), SVC_GETARGS()	Decodes the arguments of an RPC request associated with the RPC service transport handle	<rpc/svc.h>	Yes	820
svc_getcaller(), SVC_GETCALLER()	Gets the network address of the caller of the procedure associated with a RPC service transport handle	<rpc/svc.h>	Yes	821
svc_getreq()	Processes incoming RPC requests	rpcLib.a	Yes	822
svc_getreqset()	Implements custom asynchronous event processing	rpcLib.a	Yes	823

Table 29. Remote Procedure Call Support Functions

Function or Macro	Description	Library or Definition	Appl	Page
svc_register()	Maps a remote procedure to a service dispatch procedure	rpcLib.a	Yes	824
svc_run()	Waits for an RPC service request to arrive	rpcLib.a	Yes	825
svc_sendreply()	Returns the results of a remote procedure call.	rpcLib.a	Yes	826
svc_unregister()	Removes the mapping between a dispatch subroutine and a service procedure	rpcLib.a	Yes	827
svcerr_auth()	Indicates that a remote procedure call cannot be completed because of a authentication error	rpcLib.a	Yes	828
svcerr_decode()	Indicates that the parameters specified in a request cannot be decoded	rpcLib.a	Yes	829
svcerr_noproc()	Indicates that the requested procedure number does not exist	rpcLib.a	Yes	830
svcerr_noprog()	Indicates that the requested program is not registered with the RPC package	rpcLib.a	Yes	831
svcerr_progvers()	Indicates that the requested program version is not registered with the RPC package	rpcLib.a	Yes	832
svcerr_systemerr()	Indicates that a system error not covered by a protocol occurred	rpcLib.a	Yes	833
svcerr_weakauth()	Indicates that the remote call cannot be completed because of insufficient supplied authentication parameters	rpcLib.a	Yes	834
svcfcd_create()	Creates a service on any open socket descriptor	rpcLib.a	Yes	835
svccraw_create()	Creates an RPC service transport handle used for simulation	rpcLib.a	Yes	836
svctcp_create()	Creates an RPC service transport handle using TCP/IP	rpcLib.a	Yes	837
svcudp_bufcreate()	Creates an RPC service transport using UDP and supporting user-specified send and receive buffers	rpcLib.a	Yes	838
svcudp_create()	Creates an RPC service transport handle using UDP	rpcLib.a	Yes	839
xdr_accepted_reply()	Encodes an RPC reply message	rpcLib.a	Yes	962
xdr_array()	XDR filter primitive; translates between a variable-length array and its external representation	rpcLib.a	Yes	963
xdr_authunix_params()	Generates UNIX credentials	rpcLib.a	Yes	964

Table 29. Remote Procedure Call Support Functions

Function or Macro	Description	Library or Definition	Appl	Page
xdr_bool()	XDR filter primitive; translates between a C bool_t and its external representation	rpcLib.a	Yes	965
xdr_bytes()	XDR filter primitive; translates between a counted byte array and its external representation	rpcLib.a	Yes	966
xdr_callhdr()	Generates a call header similar to an RPC call header	rpcLib.a	Yes	967
xdr_callmsg()	Generates a message similar to an RPC message	rpcLib.a	Yes	968
xdr_char()	XDR filter primitive; translates between a C char and its external representation	rpcLib.a	Yes	969
xdr_destroy(), XDR_DESTROY()	Invokes the destroy function associated with a XDR stream and frees its private data structures	<rpc/xdr.h>	Yes	970
xdr_double()	XDR filter primitive; translates between a C double and its external representation	rpcLib.a	Yes	971
xdr_enum()	XDR filter primitive; translates between a C enum and its external representation	rpcLib.a	Yes	972
xdr_float()	XDR filter primitive; translates between a C float and its external representation	rpcLib.a	Yes	973
xdr_free()	Provides generic storage deallocation	rpcLib.a	Yes	974
xdr_getpos(), XDR_GETPOS()	Invokes the get-position routine associated with an XDR stream handle	<rpc/xdr.h>	Yes	975
xdr_inline(), XDR_INLINE()	Invokes the in-line routine associated with an XDR stream handle	<rpc/xdr.h>	Yes	976
xdr_int()	XDR filter primitive; translates between a C int and its external representation	rpcLib.a	Yes	977
xdr_long()	XDR filter primitive; translates between a C long and its external representation	rpcLib.a	Yes	978
xdr_opaque()	XDR filter primitive; translates between fixed-size opaque data and its external representation	rpcLib.a	Yes	979
xdr_opaque_auth()	Generates an authentication information message similar to an RPC authentication information message	rpcLib.a	Yes	980
xdr_pmap()	Describes and generates parameters for portmap procedures	rpcLib.a	Yes	981

Table 29. Remote Procedure Call Support Functions

Function or Macro	Description	Library or Definition	Appl	Page
xdr_pmaplist()	Describes and generates a list of port mappings	rpcLib.a	Yes	982
xdr_pointer()	XDR filter primitive; translates between a C pointer and its external representation; can serialize and deserialize trees.	rpcLib.a	Yes	983
xdr_reference()	XDR filter primitive; translates between a C pointer and its external representation	rpcLib.a	Yes	984
xdr_rejected_reply()	Describes RPC message rejection replies	rpcLib.a	Yes	985
xdr_replymsg()	Describes RPC message replies	rpcLib.a	Yes	986
xdr_setpos(), XDR_SETPOS()	Invokes the set-position routine associated with an XDR stream	<rpc/xdr.h>	Yes	987
xdr_short()	XDR filter primitive; translates between a C short and its external representation	rpcLib.a	Yes	988
xdr_string()	XDR filter primitive; translates between a C string and its external representation	rpcLib.a	Yes	989
xdr_u_char()	XDR filter primitive; translates between a C u_char and its external representation	rpcLib.a	Yes	990
xdr_u_int()	XDR filter primitive; translates between a C u_int and its external representation	rpcLib.a	Yes	991
xdr_u_long()	XDR filter primitive; translates between a C u_long and its external representation	rpcLib.a	Yes	992
xdr_u_short()	XDR filter primitive; translates between a C u_short and its external representation	rpcLib.a	Yes	993
xdr_union()	XDR filter primitive; translates between a discriminated C union and its external representation	rpcLib.a	Yes	994
xdr_vector()	XDR filter primitive; translates between a fixed-length array and its external representation	rpcLib.a	Yes	995
xdr_void()	XDR primitive; passed to an RPC routine requiring a dummy function parameter	rpcLib.a	Yes	996
xdr_wrapstring()	XDR primitive; passes an additional parameter to xdr_string()	rpcLib.a	Yes	997
xdrmem_create()	Initializes, in local storage, an XDR stream	rpcLib.a	Yes	998
xdrrec_create()	Provides an XDR stream that can contain long sequences of records containing data in XDR form	rpcLib.a	Yes	999

Table 29. Remote Procedure Call Support Functions

Function or Macro	Description	Library or Definition	Appl	Page
xdrrec_endofrecord()	Marks outgoing data as a record	rpcLib.a	Yes	1000
xdrrec_eof()	Checks the buffer for an input stream indicating end-of-file (EOF)	rpcLib.a	Yes	1001
xdrrec_skiprecord()	Moves the position of an input stream past the current record boundary into the beginning of the next record	rpcLib.a	Yes	1002
xdrstdio_create()	Initializes an XDR data stream	rpcLib.a	Yes	1003
xprt_register()	Registers an RPC service transport handle with the RPC program	rpcLib.a	Yes	1004
xprt_unregister()	Removes an RPC service transport handle pointed from the RPC service program	rpcLib.a	Yes	1005

Loader Functions

Table 30. Loader Functions

Function or Macro	Description	Library or Definition	Appl	Page
ld()	Loads an executable program into memory, relocates it, links it with other loaded programs, and updates the symbol table Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	352
ldrDup_name()	Returns the names of duplicate named symbols from the loaded object Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	356
ldrEntry_get()	Returns the entry address of the loaded object Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	359
ldrFree()	Frees storage used to hold the loader section of a loaded object Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	377
ldrIsexec()	Returns the executable status of a loaded object Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	360

Table 30. Loader Functions

Function or Macro	Description	Library or Definition	Appl	Page
ldr_ldinfo()	Writes information about all loaded objects into a buffer Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	374
ldrLoad()	Loads an open file into memory and relocates it to the loaded location Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	363
ldrMsg()	Displays error and information messages from loader function calls Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	368
ldrQuery()	Reads information from an opened object file Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	379
ldrResolve()	Resolves references between all loaded objects that have resident loader tables Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	370
ldrStatus()	Displays status information for all loaded objects Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	380
ldrUnload()	Unloads an object that was previously loaded by <code>ld()</code> or <code>ldrLoad()</code> Usage Note: <code>ldrLib_init()</code> must have been run first	ldrLib.a	Yes	381

Symbol Table Functions

Table 31. Symbol Table Functions

Function or Macro	Description	Library or Definition	Appl	Page
statsym_find_addr()	Searches the static symbol table for a symbol with the address closest to an address argument	symLib.a	Yes	771
statsym_find_ext()	Searches the static symbol table for an external symbol	symLib.a	Yes	772
statsym_find_ldsym()	Searches the static symbol table of a single loaded object for an external symbol	symLib.a	Yes	773

Table 31. Symbol Table Functions

Function or Macro	Description	Library or Definition	Appl	Page
statsym_find_name()	Searches the static symbol table for a symbol	symLib.a	Yes	775
statsym_find_tgaddr()	Searches the static symbol table for the symbol with the address closest to an address within the scope of symbols available to a thread group	symLib.a	No	776
statsym_find_tgname()	Searches the static symbol table for the symbol of a name within the scope of symbols available to a thread group	symLib.a	No	777
statsym_find_tgtoc()	Returns the address of the function descriptor or TOC of a symbol name	symLib.a	No	778
statsym_find_toc()	Returns the address of the function descriptor or TOC of a symbol name, if found	symLib.a	Yes	779
statsym_init()	Initializes internal tables required by the static symbol table functions	symLib.a	No	780
statsym_remove()	Removes symbols from the symbol table added using statsym_update() Usage Note: Only symbols local to calling routine's thread group may be removed	symLib.a	Yes	781
statsym_update()	Adds the symbols from an object loaded with ldrLoad() into the static symbol table Usage Note: Only symbols local to calling routine's thread group may be removed	symLib.a	Yes	782
symptr_get()	Returns the address of the static symbol table and sets a variable to the number of symbols	kadtLib.a	Yes	840

Character Handling Utilities

Table 32. Character Handling Utilities

Function or Macro	Description	Library or Definition	Appl	Page
isalnum()	Returns nonzero (true) for any character for which isalpha() or isdigit() return true	cLib.a	Yes	333
isalpha()	Returns nonzero (true) for any character for which islower() or isupper() return true	cLib.a	Yes	335
iscntrl()	Returns nonzero (true) for any control character	cLib.a	Yes	337
isdigit()	Returns nonzero (true) for any decimal digit	cLib.a	Yes	338

Table 32. Character Handling Utilities

Function or Macro	Description	Library or Definition	Appl	Page
isgraph()	Returns nonzero (true) for any printable character except the space character (' ')	cLib.a	Yes	339
islower()	Returns nonzero (true) for any lowercase letter	cLib.a	Yes	340
isprint()	Returns nonzero (true) for any printable character, including the space character (' ')	cLib.a	Yes	341
ispunct()	Returns nonzero (true) for any character for which isalnum() returns 0 (false) or that is not a space (' ') character	cLib.a	Yes	343
isspace()	Returns nonzero (true) for any standard whitespace character	cLib.a	Yes	344
isupper()	Returns nonzero (true) for any character that is an uppercase letter	cLib.a	Yes	347
isxdigit()	Returns nonzero (true) for any character that is a hexadecimal digit	cLib.a	Yes	348
mblen()	Conditionally returns 1, 0, or -1; provided for ANSI C compatibility	cLib.a	Yes	399
mbtowc()	Returns 1; provided for ANSI C compatibility	cLib.a	Yes	400
mbstowcs()	Returns the value of	cLib.a	Yes	
tolower()	Converts a character to lowercase if it represents an uppercase letter	cLib.a	Yes	906
toupper()	Converts a character to uppercase if it represents a lowercase letter	cLib.a	Yes	907
wctomb()	Returns 1; provided for ANSI C compatibility	cLib.a	Yes	958

Math Utilities

Table 33. Math Utilities

Function or Macro	Description	Library or Definition	Appl	Page
abs()	Returns the absolute value of a number	cLib.a	Yes	51
acos()	Computes the principal value of the arc cosine of a number	mathLib.a	Yes	57
asin()	Computes the principal value of the arc sine of a number	mathLib.a	Yes	66
atan()	Computes the principal value of the arc tangent of a number	mathLib.a	Yes	68

Table 33. Math Utilities

Function or Macro	Description	Library or Definition	Appl	Page
atan2()	Computes the principal value of the arc tangent of y/x	mathLib.a	Yes	68
ceil()	Computes the smallest integral value not less than a floating-point number	mathLib.a	Yes	91
cos()	Computes the cosine of a number	mathLib.a	Yes	134
cosh()	Computes the hyperbolic cosine of a number	mathLib.a	Yes	135
div()	Computes the quotient and remainder of the division of an integer numerator by an integer denominator	cLib.a	Yes	176
exp()	Computes the exponential function of a number	mathLib.a	Yes	188
fabs()	Computes the absolute value of a floating-point number	mathLib.a	Yes	189
floor()	Computes the smallest integral value not greater than a floating-point number	mathLib.a	Yes	208
fmod()	Computes the floating-point remainder of a floating-point numerator and a floating-point denominator	mathLib.a	Yes	209
frexp()	Breaks a floating-point number into a normalized fraction and an integral power of two	mathLib.a	Yes	230
labs()	Computes the absolute value of a long integer	cLib.a	Yes	351
ldexp()	Multiplies a floating point number by two raised to a power	mathLib.a	Yes	354
ldiv()	Computes the quotient and remainder of the division of a long integer numerator by a long integer denominator	cLib.a	Yes	355
log()	Computes the natural logarithm of a number	mathLib.a	Yes	390
log10()	Computes the base-ten logarithm of a number	mathLib.a	Yes	391
modf()	Breaks a floating-point number into integral and fractional parts	mathLib.a	Yes	434
pow()	Computes x raised to the power y	mathLib.a	Yes	499
rand()	Generates a pseudo-random integer Usage Note: Use rand_r()	cLib.a	No	631
rand_r()	Reentrant version of rand()	cLib.a	Yes	632
sin()	Computes the sine of a number	mathLib.a	Yes	753
sine()	Computes the hyperbolic sine of a number	mathLib.a	Yes	754

Table 33. Math Utilities

Function or Macro	Description	Library or Definition	Appl	Page
sqrt()	Computes the non-negative square root of a number	mathLib.a	Yes	764
srand()	Sets the seed for producing a series of pseudo-random integers Usage Note: Use rand_r() and maintain a seed	cLib.a	No	765
tan()	Computes the tangent of a number	mathLib.a	Yes	847
tanh()	Computes the hyperbolic tangent of a number	mathLib.a	Yes	848

Queue Utilities

Table 34. Queue Utilities

Function or Macro	Description	Library or Definition	Appl	Page
queCreate()	Creates queue data types (queNode_t) and adds the queue data type to the root queue data type	queLib.a	Yes	617
queDeq()	Removes the oldest node from a queue and returns the address of the removed node	queLib.a	Yes	619
queDeqNoCheck()	Removes the oldest node from a queue and returns the address of the removed node; does not check whether the queue is empty.	queLib.a	Yes	620
queEnq()	Places a node at the end of a queue	queLib.a	Yes	621
queEnqFront()	Places a node at the front of a queue	queLib.a	Yes	622
queInit()	Initializes the root queue data type in a doubly linked list of queue nodes	queLib.a	Yes	623
queInsert()	Inserts a node in a queue following a specified node	queLib.a	Yes	624
queNewer()	Returns the address of the next node in a queue that is newer than a specified node	queLib.a	Yes	625
queNewest()	Returns the address of the newest node in a queue	queLib.a	Yes	626
queOlder()	Returns the address of the next node in a queue that is older than a specified node	queLib.a	Yes	627
queOldest()	Returns the address of the oldest node in a queue	queLib.a	Yes	628
queRemove()	Removes a node from its queue	queLib.a	Yes	629

Ring Buffer Utilities

Table 35. Ring Buffer Utilities

Function or Macro	Description	Library or Definition	Appl	Page
rngBufGet()	Retrieves characters from a ring buffer and stores them in a buffer	rngLib.a	Yes	662
rngBufPut()	Stores characters from a buffer in a ring buffer	rngLib.a	Yes	663
rngCount()	Determines the number of characters in a ring buffer	rngLib.a	Yes	664
rngCreate()	Creates a ring buffer	rngLib.a	Yes	665
rngDelete()	Destroys a ring buffer	rngLib.a	Yes	666
rngFlush()	Empties a ring buffer	rngLib.a	Yes	667
rngIsEmpty()	Returns a nonzero value if the ring buffer is empty or 0 if it is not	rngLib.a	Yes	668
rngIsFull()	Returns a nonzero value if the ring buffer is full, or 0 if it is not	rngLib.a	Yes	669

String Handling Utilities

Table 36. String Handling Utilities

Function or Macro	Description	Library or Definition	Appl	Page
mbstowcs()	Returns the value of one of its arguments; provided for ANSI C compatibility	cLib.a	Yes	401
strcasecmp()	Compares a string to another string, ignoring case	cLib.a	Yes	785
strcat()	Appends a copy of a string (including the terminating NULL character) to the end of another string	cLib.a	Yes	786
strchr()	Locates the first occurrence of an integer (converted to a char) in a string	cLib.a	Yes	787
strcmp()	Compares a string to another string	cLib.a	Yes	790
strcoll()	Calls strcmp()	<string.h>	Yes	791
strcpy()	Copies a string (including the terminating NULL character) into an array	cLib.a	Yes	792
strcspn()	Computes the length of the maximum initial segment of a string that consists entirely of characters not found in another string	cLib.a	Yes	793

Table 36. String Handling Utilities

Function or Macro	Description	Library or Definition	Appl	Page
strerror()	Maps an error number to an error message string and returns a pointer to the string	cLib.a	Yes	794
strlen()	Computes the length of a string	cLib.a	Yes	797
strncasecmp()	Compares at most <i>n</i> characters of a string to another string, ignoring case	cLib.a	Yes	798
strncat()	Appends at most <i>n</i> characters from an array to the end of a string	cLib.a	Yes	799
strncmp()	Compares at most <i>n</i> characters of a string to another string	cLib.a	Yes	800
strncpy()	Copies at most <i>n</i> characters of a string into an array	cLib.a	Yes	801
strpbrk()	Locates the first occurrence in a string of any character found in a string of delimiters	cLib.a	Yes	802
strrchr()	Locates the last occurrence of an integer (converted to a char) in a string	cLib.a	Yes	803
strspn()	Computes the length of the maximum initial segment of a string that consists entirely of characters from another string	cLib.a	Yes	804
strstr()	Locates the first occurrence in a string of a sequence of characters in another string	cLib.a	Yes	805
strtok()	A sequence of calls to strtok() decomposes a string into a sequence of tokens, each of which is delimited by a character from another string Usage Note: Use strtok_r()	cLib.a	No	808
strtok_r()	Reentrant version of strtok()	cLib.a	Yes	811
strtol()	Converts a character string to a sequence of characters, each of which can be interpreted as a long int	cLib.a	Yes	813
strtoul()	Converts a character string to a sequence of characters, each of which can be interpreted as an unsigned long int	cLib.a	Yes	815
strxfrm()	Calls strncpy()	<string.h>	Yes	817
wcstombs()	Returns the value of one of its arguments; provided for ANSI C compatibility	cLib.a	Yes	959

General Programming Utilities

Table 37. General Programming Utilities

Function or Macro	Description	Library or Definition	Appl	Page
align_h()	Handles load or store alignment errors in PowerPC 6xx processors Usage Note: Requires installation as an event	alignLib.a	No	60
atof()	Converts the initial portion of a character string to double	cLib.a	Yes	71
atoi()	Converts a character string to an integer	cLib.a	Yes	72
atol()	Converts a character string to a long integer	cLib.a	Yes	74
bsearch()	Performs a binary search of a sorted array	cLib.a	Yes	86
cpp_exit()	xcoff only: Removes the C++ runtime environment and calls static constructors defined in load image	cpprtLib.a	Yes	136
cpp_init()	xcoff only: Initializes the C++ runtime environment and calls static destructors defined in load image	cpprtLib.a	Yes	137
dcache_flush()	Flushes cache lines, beginning at effective address and continuing for <i>count</i> bytes	ioLib.a	Yes	168
dcache_invalidate()	Invalidates cache lines, beginning at effective address and continuing for <i>count</i> bytes	ioLib.a	No	169
getenv()	Searches the environment variable table for an entry containing a value Usage Note: Environment space is global	cLib.a	Yes	262
getopt()	Places <i>nargv</i> index of next argument to be processed in <i>optind</i>	runLib.a	Yes	275
getpid()	Returns the process ID of the calling process	rtxLib.a	Yes	278
inbyte()	Returns a data byte read from an I/O port	ioLib.a	Yes	307
inshort()	Returns a data halfword read from an I/O port	ioLib.a	Yes	319
inword()	Returns a data word read from an I/O port	ioLib.a	Yes	331
localeconv()	Returns a pointer to the locale structure; OS Open supports only the C locale Usage Note: Writes to the lconv struct will cause addressing exceptions	cLib.a	Yes	387
longjmp()	Restores the environment saved by a call to setjmp()	cLib.a	Yes	392

Table 37. General Programming Utilities

Function or Macro	Description	Library or Definition	Appl	Page
memchr()	Locates the first occurrence of an integer (converted to an unsigned char) in the first <i>n</i> characters (interpreted as unsigned char) of an object	cLib.a	Yes	402
memcmp()	Compares the first <i>n</i> characters of an object and the first <i>n</i> characters of another object	cLib.a	Yes	403
memcpy()	Copies characters from an object to another object	cLib.a	Yes	404
memmove()	Copies <i>n</i> characters from an object to another object; can handle overlapping memory segments	cLib.a	Yes	415
memset()	Copies the value of <i>c</i> (converted to an unsigned char) into the first <i>n</i> characters of an object	cLib.a	Yes	423
np_compare_swap()	Compares two values and conditionally replaces a value	rtxLib.a	Yes	477
outbyte()	Writes a specified data byte to an I/O port	ioLib.a	Yes	481
outshort()	Writes a specified data halfword to an I/O port	ioLib.a	Yes	482
outword()	Writes a specified data word to an I/O port	ioLib.a	Yes	483
package_install()	Provides a standard way to associate a name with a pointer	devLib.a	Yes	484
perror()	Prints an error message to the standard error	fsLib.a	Yes	488
qsort()	Sorts an array in place	cLib.a	Yes	616
run()	Loads and executes an OS Open object module	runLib.a	No	679
setenv_np()	Defines a new environment variable or redefines an existing environment variable	cLib.a	No	721
setjmp()	Copies the current execution context for potential restoration using longjmp()	cLib.a	Yes	722
setlocale()	Queries all or portions of the locale; only the C locale is supported	cLib.a	Yes	723
siglongjmp()	Restores the environment saved by a call to sigsetjmp()	rtxLib.a	Yes	745
sigsetjmp()	Copies the current execution context for potential restoration by siglongjmp()	rtxLib.a	Yes	750
tzset()	Uses the environment variable TZ to set the external variable tzname	cLib.a	No	918

Table 37. General Programming Utilities

Function or Macro	Description	Library or Definition	Appl	Page
va_arg()	Used with va_start() and va_end() to process variable argument lists	cLib.a	Yes	926
va_end()	Used with va_start() and va_arg() to process variable argument lists	cLib.a	Yes	928
va_start()	Used with va_arg() and va_end() to process variable argument lists	cLib.a	Yes	929

Debugging and Reporting Functions

Table 38. Debugging and Reporting Functions

Function or Macro	Description	Library or Definition	Appl	Page
cond_dump()	Provides information about a conditional variable	kadtLib.a	No	129
cond_list()	Provides formatted information about all condition variables	kadtLib.a	No	130
dbbrkpt_clr()	Removes one or more breakpoints	dbLib.a	No	151
dbbrkpt_cont()	Continues after encountering a breakpoint	dbLib.a	No	152
dbbrkpt_disp()	Displays breakpoint status information	dbLib.a	No	153
dbbrkpt_find_inst()	Returns the instruction at the address of a breakpoint	dbLib.a	No	154
dbbrkpt_init()	Initializes breakpoint data and installs the Trap and Error first level interrupt handlers (FLIHs)	dbLib.a	No	155
dbbrkpt_isbp()	Returns true if a permanent breakpoint is set for a thread at an address	dbLib.a	No	156
dbbrkpt_next_inst()	Determines the location that a thread will execute next and copies that location into an object	dbLib.a	No	157
dbbrkpt_set()	Sets a breakpoint for a thread at an address	dbLib.a	No	159
dbbrkpt_tclr()	Clears a temporary breakpoint	dbLib.a	No	160
dbbrkpt_tset()	Sets a temporary breakpoint	dbLib.a	No	161
dbbrkpt_write_inst()	Writes an instruction to an address and synchronizes the instruction and data caches	dbLib.a	No	162
dbmem_disp()	Displays memory starting at an address	dbLib.a	No	163

Table 38. Debugging and Reporting Functions

Function or Macro	Description	Library or Definition	Appl	Page
dbmem_listi()	Disassembles memory starting at an address for a specified number of instructions	dbLib.a	No	164
dbstepi()	Steps a thread by a specified number of instructions	dbLib.a	No	165
dbsymname_find()	Copies the name of a CSECT into a buffer and returns the offset from beginning of the CSECT	dbLib.a	No	166
dbwhere()	Displays stack trace-back information that shows the sequence of calls made by a thread	dbLib.a	No	167
flih_list()	Provides formatted information about all FLIHs	kadtLib.a	No	203
fpreg_dump()	Displays floating-point registers from a specified thread	kadtLib.a	No	219
heap_list()	Provides formatted information about all heaps	kadtLib.a	No	302
ispthreadid()	Returns nonzero (true) if the argument is a valid thread ID	kadtLib.a	No	342
issuspended()	Returns nonzero (true) if a thread is suspended at a breakpoint or an error	kadtLib.a	No	345
kda_dump()	Formats and displays the kernel data area (KDA)	kadtLib.a	No	349
library_list()	Provides formatted information about libraries registered using package_install()	kadtLib.a	No	382
mq_dump()	Provides information about a message queue	kadtLib.a	No	436
mq_list()	Provides formatted information about all message queues	kadtLib.a	No	438
mutex_dump()	Displays information about a mutex	kadtLib.a	No	462
mutex_list()	Provide formatted information about all mutexes	kadtLib.a	No	463
mutexattr_dump()	Displays information about a mutex attribute	kadtLib.a	No	465
pool_list()	Provides formatted information about all memory pools	kadtLib.a	No	497
reg_get()	Returns the contents of a register for a thread	kadtLib.a	No	647
reg_set()	Writes data into a register for a thread	kadtLib.a	No	648

Table 38. Debugging and Reporting Functions

Function or Macro	Description	Library or Definition	Appl	Page
rsld_setschedparam()	Sets scheduling policies and parameters for new threads created by a remote source level debugger	rsldLib.a	No	677
rsld_start()	Creates the remote source-level debugger daemon thread	rsldLib.a	No	678
semaphore_dump()	Displays information about a semaphore	kadtLib.a	No	710
semaphore_list()	Provides formatted information about all semaphores	kadtLib.a	No	711
signal_list()	Provides formatted information about system signals	kadtLib.a	No	747
thread_attr_dump()	Provides information about a thread attributes object	kadtLib.a	No	879
thread_debug()	Creates a thread with a breakpoint	dbLib.a	No	880
thread_dump()	Provides information about a thread	kadtLib.a	No	881
thread_info_list()	Provides formatted information about active threads	kadtLib.a	No	883
thread_list()	Lists all threads active in OS Open	kadtLib.a	No	884
thread_name()	Copies the name of a thread's starting function into a buffer	kadtLib.a	No	885
timer_dump()	Provides information about a timer	kadtLib.a	No	891
timer_list()	Provides formatted information about all timers	kadtLib.a	No	895
trace_checkall()	Scans entire kernel trace buffer for entries associated with a specified system call	dbLib.a	No	908
trace_checklast()	Scans kernel trace buffer for last entry associated with a specified system call	dbLib.a	No	909
trace_copy()	Writes kernel trace buffer to a file	dbLib.a	No	910
trace_get()	Retrieves trace information from the OS Open trace buffer	rtxLib.a	No	911
trace_snapshot()	Provides a snapshot of the OS Open trace buffer	dbLib.a	No	912
trace_write()	Writes trace information into the OS Open trace buffer	rtxLib.a	No	913

OpenShell Functions

Table 39. OpenShell Functions

Function or Macro	Description		Appl	Page
getprompt()	Returns the location of the OpenShell prompt string	shell.o	No	279
setprompt()	Sets the OpenShell prompt	shell.o	No	724
shell()	The main entry point for OpenShell Usage Note: Shell is designed to run from initial thread group	shell.o	No	732
shell_threadid()	Returns the thread ID of the OpenShell thread	shell.o	No	733

Chapter 2. OS Open Function and Macro Descriptions

This chapter contains descriptions of the OS Open functions and macros.

The function calls and macros are sorted alphabetically by name.

All descriptions contains the following sections:

- Synopsis
- Library
- Description
- Errors
- Attributes

Examples and references are provided or referenced where appropriate.

Attributes and Threads

Functions and macros have attributes that affect thread execution. Depending on their behavior, threads may or may not be “async safe,” “cancel safe,” and “interrupt handler safe.”

Async Safe Functions

An async safe function may be entered by two or more concurrently executing threads, with each thread getting the correct results.

Functions that operate only on disjoint or local data objects are reentrant, and are therefore async safe. For example, **abs()** operates only on its arguments, making it reentrant and therefore async safe.

Functions that operate on common or global data objects may use serialization techniques, such as mutexes, within the functions to ensure async safe operation. The **getchar()** and **putchar()** macros, which operate on common files, use serialization (**flockfile()** and **funlockfile()**) to achieve async safe operation.

Following is an alphabetical list of the OS Open functions which are **not** async safe:

Table 40. Functions Not Async Safe

Function and Page	Function and Page
asctime() , page 62	nfs_mount() , page 474
ctime() , page 149	nfs_showmount() , page 475
dev_io_init() , page 171	opendir() , page 480
enet_attach() , page 183	ping() , page 489
ftp() , page 241	putc_unlocked() , page 611
ftpd_start() , page 249	putchar_unlocked() , page 613
getc_unlocked() , page 256	rand() , page 631
getchar_unlocked() , page 258	readdir() , page 636
gethostbyaddr() , page 263	rms_delq() , page 655
gethostbyname() , page 265	rms_init() , page 656
getnetbyaddr() , page 270	rms_insq() , page 657
getnetbyname() , page 272	rms_oldest() , page 658
getopt() , page 275	rms_start() , page 659
getprotobyname() , page 280	rsld_start() , page 678
getprotobynumber() , page 281	run() , page 679
getservbyname() , page 290	setsysconf() , page 728
getservbyport() , page 291	setuname() , page 729
gmtime() , page 299	shell() , page 732
ifconfig() , page 304	slip_attach() , page 756
inet_ntoa() , page 316	strtok() , page 808
localtime() , page 388	tcpip_init() , page 857
net_init() , page 468	tftp() , page 870
nfs_authget() , page 471	tn() , page 902
nfs_authset() , page 472	

Cancel Safe Functions

The cancel safe attribute is important only to threads executing in deferred cancelability mode (the cancel state is enabled; the cancel type is deferred).

A thread executing in deferred cancelability mode can execute a cancel safe function without being canceled. If the same thread executes a non-cancel safe function, the thread may or may not be canceled during execution of the function.

Interrupt Handler Safe Functions

An interrupt handler safe function may be called by a first level interrupt handler (FLIH).

Character Sets Used in Function and Macro Descriptions

The function and macro descriptions refer to the following character sets:

Decimal Digits

The following characters comprise the decimal digit character set:

0 1 2 3 4 5 6 7 8 9

Graphics Characters

The following characters comprise the graphics character set:

! " # % & ' () * + , - . / : ; < = > ? [\] ^ _ { | } ~

Hexadecimal Digits

The following characters comprise the hexadecimal digit character set:

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

Lowercase Letters

The following characters comprise the lowercase letter character set:

a b c d e f g h i j k l m n o p q r s t u v w x y z

Uppercase Letters

The following characters comprise the uppercase letter character set:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Portable File Name Characters

The following characters comprise the portable file name character set:

- All uppercase letters
- All lowercase letters
- All decimal digits
- The period (.)
- The underscore (_)
- The hyphen (-)

The slash (/) delimits components in a portable path name.

abort()

Synopsis

```
#include <stdlib.h>

void abort( void );
```

Library

cLib.a

Description

abort() raises the **SIGABRT** signal and then abnormally terminates OS Open.

abort() never returns to the calling thread.

A context switch may occur between raising **SIGABRT** and abnormal termination. If a higher priority thread is blocked waiting for **SIGABRT**, that thread can run. Abnormal termination occurs when the thread that called **abort()** resumes. OS Open is terminated with a panic code of zero, which is unique to **abort()**.

Errors

None.

Example

See **read()**, p. 633.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.4.1
- **raise()**, p. 630
- **sigwait()**, p. 751

Synopsis

```
#include <stdlib.h>
int abs( int n );
```

Library

cLib.a

Description

abs() returns the absolute value of an integer argument *n*.

The result is undefined when the absolute value of the argument cannot be represented as an integer. The minimum value of *n* is defined by INT_MIN + 1 in the **<limits.h>** include file. For example, on 32 bit processors, the absolute value of -2147483648 cannot be represented as an integer.

Errors

None.

Example

See **atoi()**, p. 72.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.6.1
- **labs()**, p. 351

accept()

Synopsis

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

int accept( int s, struct sockaddr *addr, int *addrlen );
```

Library

tcpipLib.a

Description

accept() accepts a connection on a socket and returns a new socket for the connection.

This call is used with the connection-based socket type **SOCK_STREAM**.

The argument *s* names a socket created using **socket()**, bound to an address using **bind()**, and listening for connections after **listen()**. **accept()** extracts the first connection request in the queue of pending connections, creates a socket with the properties of *s*, and allocates a new file descriptor for the new socket.

If no pending connections are in the queue, and the socket is not marked as non-blocking, **accept()** blocks the caller until a connection is present. If the socket is marked as non-blocking and no pending connections are in the queue, **accept()** returns an error.

The accepted socket cannot accept more connections, and *s* remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity known to the communication layer. The communication domain determines the format of *addr*.

The argument *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*. On its return, *addrlen* contains the length of the returned address.

If successful, **accept()** returns a non-negative integer file descriptor. Otherwise, **accept()** returns `-1` and sets *errno*.

Errors

[EBADF]	The descriptor <i>s</i> is not valid.
[EINVAL]	accept() not allowed for <i>s</i> .
[ENOTSOCK]	The descriptor <i>s</i> references a file, not a socket.
[EOPNOTSUPP]	<i>s</i> is not of type SOCK_STREAM .
[EWOULDBLOCK]	<i>s</i> is marked non-blocking and no connections are present to be accepted.

Example

The following example accepts new connections on an existing socket:

```
#include <stdio.h>
#include <types.h>
#include <netinet/in.h>
#include <sys/select.h>

void * ex_accept(int sd) {
    int          maxsock; /* Max # of sockets to select on */
    int * allsock, * readable; /* File descriptor sets */
    int          fdsetsize; /* Number bytes in fd set */
    int          lsd; /* New socket descriptor */
    struct sockaddr_in lsa; /* Used in accept() call */
    int ia = sizeof lsa.sin_addr; /* " " " " */
    int          n; /* # of sockets with data waiting */
    int          rc;

    fdsetsize=32;
    allsock=(int *)malloc(fdsetsize);
    if (NULL==allsock)
        return((void *)-1);

    readable=(int *)malloc(fdsetsize);
    if (NULL==readable) {
        free(allsock);
        return((void *)-1);
    }

    memset(allsock,0,fdsetsize);
    maxsock=8;
    FD_SET(sd,allsock); /* Add socket to list for select */

    for(;;) { /* Forever */
        memcpy(readable,allsock,fdsetsize);
        n = select(maxsock, readable, NULL, NULL, NULL);
        if (0==n) {
            printf("select() returned 0\n");
            continue;
        }

        if ( (lsd=accept(sd,(struct sockaddr *)&lsa,&ia)) < 0) {
            printf("accept() failed\n");
            return((void *)-1);
        }
    }
}
```

accept()

```
    printf("received a connection\n");

} /* end for Forever */
/* NOTREACHED*/
} /* end */
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.3
- **bind()**, p. 80
- **connect()**, p. 132
- **listen()**, p. 386
- **select()**, p. 691
- **socket()**, p. 760

Synopsis

```
#include <unistd.h>

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

Library

devLib.a

Description

access() checks the file pointed to by *path* for the access permissions specified by *amode*.

amode should be set to either a simple existence check, **F_OK**, or to the bitwise inclusive OR any combination of **R_OK** (read permission), **W_OK** (write permission), and **X_OK** (permission to execute).

If the requested access is permitted, **access()** returns 0. Otherwise, **access()** returns -1 and sets *errno*.

Errors

[EACCES]	The permissions specified by <i>amode</i> are denied, or search permission is denied for a path name component.
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of <i>path</i> exceeds {PATH_MAX} .
[ENOENT]	<i>path</i> names a file that does not exist or points to an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.

Example

The following example demonstrates several regular file functions.

```
#include <unistd.h>

int reg_file(char *fname)
{
    int rc;
    int fd;
    struct stat stat_buf;
    char new_name[40];

    rc = access(fname, F_OK); /* does file exist? */
    if (rc != 0) {
        perror("access failed");
        return(rc);
    }
    /* now obtain file statistics */
```

access()

```
rc = stat(fname, &stat_buf);
if (rc != 0) {
    perror("stat call");
    return(rc);}
/* print out some of the file statistics */
printf("file mode: %x file size: %d number of links: %d\n",
       stat_buf.st_mode, stat_buf.st_size, stat_buf.st_nlink);

/* now, rename the file */
strcpy(new_name, fname); /* copy old name */
strcat(new_name, ".old"); /* add suffix of "old" */
rc = rename(fname, new_name); /* rename file */
if (rc != 0) {
    perror("rename call");
    return(rc);}
/* open file by new name */
fd = open(new_name, O_RDWR);
if (fd == -1) {
    perror("open call");
    return(-1);}
/* get stats by file descriptor */
rc = fstat(fd, &stat_buf);
if (rc != 0) {
    perror("fstat call");
    return(rc);}
/* print out some of the file statistics */
printf(" for file desc. %d, file mode: %x file size: %d number of links: %d\n",
       fd, stat_buf.st_mode, stat_buf.st_size, stat_buf.st_nlink);
/* finally , unlink file */
close(fd);
rc = unlink(new_name);
if (rc != 0) {
    perror("unlink call");
    return(rc);
}
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.6.3

Synopsis

```
#include <math.h>
double acos( double x );
```

Library

mathLib.a

Description

acos() computes the principal value of the arc cosine of x .

The returned arc cosine is in the range $[0, \pi]$ radians.

If x is not in the range $[-1, +1]$, a domain error occurs and **acos()** sets *errno*.

Errors

[EDOM] x is outside the domain of the acos function.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.2.1

alarm()

Synopsis

```
#include <unistd.h>

unsigned int alarm( unsigned int seconds );
```

Library

rtxLib.a

Description

alarm() instructs OS Open to deliver the signal **SIGALRM** to the system process (OS Open) after *seconds* real time seconds elapse.

Alarm requests are not stacked. Only one request for **SIGALRM** can be outstanding. See **timer_create()** on page 888 for other ways to generate signals after a specified period.

If **SIGALRM** was not generated at the time of the call (**alarm()** has not elapsed), a subsequent call to **alarm()** reschedules **SIGALRM** generation. If *seconds* is 0, any outstanding **alarm()** calls for OS Open are canceled.

If a previous **alarm()** call had time remaining, **alarm()** returns the number of seconds until the function would have generated **SIGALRM**. Otherwise, **alarm()** returns 0.

Errors

None.

Example

The following example waits 15 seconds for an attention signal. If a **SIGINT** signal is received, or if 15 seconds pass, a global message queue sends a message to the parent thread. The thread is terminated when the parent thread sends a **SIGKILL** signal.

```
#include <signal.h>
#include <unistd.h>
#include <mqueue.h>
#include <pthread.h>

typedef enum {TRAP_MSG, ERR_MSG, ATTN_MSG} brkpt_msgtype_t;
typedef struct dbbrkpt_msg {
    brkpt_msgtype_t msgtype;
    pthread_t thread;
} dbbrkpt_msg_t;

extern mqd_t mqdes1;

void *attn_thread(void *arg) {
```

```
dbbrkpt_msg_t msg;
sigset_t set;
int sig;
int rc=0;

sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGKILL);
sigaddset(&set, SIGALRM);
alarm(15);          /* Set alarm for 15 seconds */
if ((rc=sigwait(&set, &sig)) ==0) {
    if (sig == SIGINT || sig == SIGALRM) {
        msg.msgtype = ATTN_MSG;
        msg.thread = pthread_self();
        mq_send(mqdes1, (char *)&msg, sizeof(msg), 15);
    }
    if (sig != SIGALRM)
        alarm(0);      /* Cancel alarm */
}
else
    printf("sigwait returned an error\n")
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §3.4.1

align_h()

Synopsis

```
#include <alignLib.h>
void *align_h(int *counter);
```

Library

alignLib.a

Description

align_h() is used to handle alignment errors in 6xx processors. These errors occur when load or store operations are used to access data not on the “natural” address boundary for the operation. Most of the time, this situation is handled transparently by the hardware. There are other cases, especially when data relocation is active, that the alignment exception is generated and must be handled by software. The processor User’s Manual describes the conditions under which alignment exceptions occur. The **align_h()** function will correctly emulate the misaligned operation and is installed as an exception handler via **int_install()**. The handler requires ALIGN_STACK_SIZE bytes of working stack.

If the exception argument is non NULL, it is treated as the address of a word that will be incremented every time the handler executes. This feature can be used to count instances of misaligned data access in an application.

Errors

None.

Example

The following shows one method of installing the alignment exception handler:

```
#include <alignLib.h>
int align_counter = 0;
flih_t a_flih;
char *tstack;

tstack = (char *)malloc(ALIGN_STACK_SIZE);
if (tstack == NULL) return(-1);
/* register interrupt handler */
a_flih.flih_stack = (void *) (tstack + ALIGN_STACK_SIZE);
a_flih.flih_function = (void *) align_h;
a_flih.arg = (void *) &align_counter;
if (int_install(EVENT_ALIGNMENT_NP, &a_flih, NULL) != 0) return(-1);
/* align_counter will count the number of software handled
alignment exceptions */
```

Attributes

Async Safe	n/a
Cancel Safe	n/a
Interrupt Handler Safe	n/a

References

- `int_install()`, p. 327

asctime()

Synopsis

```
#include <time.h>
char *asctime( const struct tm *time );
```

Library

cLib.a

Description

asctime() converts time, stored as a structure pointed to by *time*, to a character string and returns a pointer to the resulting string.

time can be obtained using **gmtime()**, **localtime()**, **gmtime_r()**, or **localtime_r()**; these functions return a pointer to a *tm* structure, defined in **<time.h>**. (See **gmtime()** on page 299 for a description of the *tm* fields.)

The string produced by **asctime()** is converted into a 26-character string with constant-width fields as illustrated in the following example:

```
Sat Jul 14 02:03:55 1987\n\n0
```

asctime() uses a 24-hour-clock. Days and months are abbreviated to their first three letters, beginning with a capital. Newline (`\n`) and NULL (`\0`) end the string.

Note: **asctime()** and **ctime()** each store results in a shared statically allocated buffer. A call to either function destroys the result of a previous call.

Errors

None.

Example

The following example prints local time and universal coordinated time.

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

void time_stamp(void)
{
    struct tm *t;
    char *prt_time;
    time_t seconds;
    /* Get the number of seconds since 01/01/70 00:00:00 */
    time(&seconds);
    /* Break out the time to universal coordinated time */
    t = gmtime(&seconds);
    prt_time = asctime(t);
    puts(prt_time);
}
```

```
/* Break out time to local time */  
t = localtime(&seconds);  
prt_time = asctime(t);  
puts(prt_time);  
}
```

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.12.3.1
- **asctime_r()**, p. 64
- **ctime_r()**, p. 150
- **gmtime()**, p. 299
- **localtime()**, p. 388
- **time()**, p. 886

asctime_r()

Synopsis

```
#include <time.h>
char *asctime_r( const struct tm *time, char *buf );
```

Library

cLib.a

Description

asctime_r() is a reentrant version of **asctime()**.

Unlike **asctime()**, the resulting string is not stored in a statically allocated buffer. Instead, the string is stored in a user-provided buffer, pointed to by *buf*, that must contain space for at least 26 characters.

If successful, **asctime_r()** returns a pointer to the string result that is equivalent to *buf*. Otherwise, **asctime_r()** returns NULL.

Errors

None.

Example

The following example prints local and universal coordinated time (UTC). Unlike the example in **asctime()**, the following example is reentrant.

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

void time_stamp(void)
{
    struct tm t;
    char prt_time[26];
    time_t seconds;
    /* Get the number of seconds since 01/01/70 00:00:00 UTC*/
    time(&seconds);
    /* Break out the time to universal coordinated time */
    gmtime_r(&seconds, &t);
    asctime_r(&t, prt_time);
    puts(prt_time);
    /* Break out time to local time */
    localtime_r(&seconds, &t);
    asctime(&t, prt_time);
    puts(prt_time);
}
```


Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §19.2.3.4
- **asctime()**, p. 62

asin()

Synopsis

```
#include <math.h>
double asin( double x );
```

Library

mathLib.a

Description

asin() computes the principal value of the arc sine of x .

The returned arc sine is in the range $[-\pi/2, +\pi/2]$ radians.

If x is not in the range $[-1, +1]$, a domain error occurs and **asin()** sets *errno*.

Errors

[EDOM] x is outside the domain of the asin function.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.2.2

Synopsis

```
#include <assert.h>

void assert( int expression );
```

Library

None. This is a macro defined in **<assert.h>**.

Description

assert() writes information to the standard error if *expression* is false and then calls **abort()**.

The information includes the text of the argument, the source file name and the source line number.

To remove the effects of the **assert()** macro, define the constant **NDEBUG** before including the file **<assert.h>**.

Note: The clock named by **CLOCK_REALTIME** must be running and the file system must have been initialized by **fs_init()**.

assert() returns nothing.

Errors

None.

Example

See **fs_init()**, p. 231.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.2.1
- **abort()**, p. 50

atan()

Synopsis

```
#include <math.h>
double atan( double x );
```

Library

mathLib.a

Description

atan() computes the principal value of the arc tangent of x .
The returned arc tangent is in the range $[-\pi/2, +\pi/2]$ radians.

Errors

[ERANGE]	The result is out of range.
----------	-----------------------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.2.3

Synopsis

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

Library

mathLib.a

Description

atan2() computes the principal value of the arc tangent of y/x , using the signs of both arguments to determine the quadrant of the returned value.

The returned arc tangent of y/x is in the range $[-\pi, +\pi]$ radians.

If both arguments are 0, a domain error occurs and **atan2()** sets *errno*.

Errors

[EDOM]	x and y are 0.
[ERANGE]	The result is out of range.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.2.4

atexit()

Synopsis

```
#include <stdlib.h>
int *atexit( void (*func) (void) );
```

Library

cLib.a

Description

atexit() registers the function pointed to by *func* to be called without arguments at normal thread termination.

atexit() can register up to 32 functions.

If successful, **atexit()** returns 0. Otherwise, **atexit()** returns a non-zero value.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.10.4.2
- **exit()**, p. 187
- **pthread_exit()**, p. 557

Synopsis

```
#include <stdlib.h>

double atof( const char *nptr );
```

Library

cLib.a

Description

atof() converts the initial portion of the string pointed to by *nptr* to **double** representation and returns the converted value.

Except for the behavior of **strtod()** on error, **atof()** is equivalent to:

```
strtod(nptr, (char **) NULL)
```

Note: **atof()** should be called only by threads enabled for floating-point arithmetic.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.1.1
- **strtod()**, p. 806

atoi()

Synopsis

```
#include <stdlib.h>

int atoi( const char *string );
```

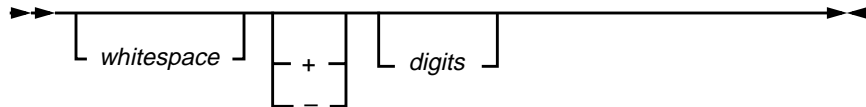
Library

cLib.a

Description

atoi() converts a character string to an integer.

The input *string* is a sequence of characters, in the following form, that can be interpreted as an integer:



whitespace consists of characters for which **isspace()** is true, such as spaces and tabs. **atoi()** ignores leading whitespace characters.

The *digits* consists of one or more characters for which **isdigit()** is true.

atoi() returns an integer produced by interpreting the input characters as a number.

atoi() stops reading the input string at the first character that it cannot recognize as part of a number. This character can be a NULL that ends the string.

If **atoi()** cannot convert the input to an integer, the function returns 0. If an overflow occurs, the return value is undefined.

Errors

None.

Example

The following example converts a string to a decimal integer and returns its absolute value, rounded to the nearest tens digit.

```
#include <stdlib.h>

int conv_int(char *string)
{
    div_t round;
    int number;
    /* Convert the string to a decimal integer */
    number = atoi(string);
```



```
/* Round number to the nearest tens digit */
round = div(number, 10);
if(round.rem >= 5)
    number = round.quot + 1;
else if (round.rem <= -5)
    number = round.quot - 1;
else
    number = round.quot;
number *= 10;
/* Get the absolute value of the number */
number = abs(number);
return(number);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.1.2

atol()

Synopsis

```
#include <stdlib.h>
long int atol( const char *string );
```

Library

cLib.a

Description

atol() converts a character string to a long integer.

The input *string* is a sequence of characters, in the following form, that can be interpreted as an integer



whitespace consists of characters for which **isspace()** is true, such as spaces and tabs. **atol()** ignores leading whitespace characters.

The *digits* consists of one or more characters for which **isdigit()** is true.

atol() returns a long integer produced by interpreting the input characters as a number.

atol() stops reading the input string at the first character that it cannot recognize as part of a number. This character can be a NULL that ends the string.

If **atol()** cannot convert the input to an integer, the function returns 0. If an overflow occurs, the return value is undefined.

Errors

None.

Example

The following example converts a string to a decimal long and returns its absolute value, rounded to the nearest tens digit.

```
#include <stdlib.h>

long conv_long(char *string)
{
    ldiv_t round;
    long number;
    /* Convert the string to a decimal integer */
```

```

number = atol(string);
/* Round number to the nearest tens digit */
round = ldiv(number, 10);
if(round.rem >= 5)
    number = round.quot + 1;
else if (round.rem <= -5)
    number = round.quot - 1;
else
    number = round.quot;
number *= 10;
/* Get the absolute value of the number */
number = labs(number);
return(number);
}

```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.1.3

auth_destroy(), AUTH_DESTROY()

Synopsis

```
#include <rpc/rpc.h>
void auth_destroy( auth *auth )
```

Library

None. This is a macro defined in `<rpc/auth.h>`

Description

auth_destroy(), also called **AUTH_DESTROY()**, destroys the authentication information structure pointed to by *auth*.

Destroying the structure deallocates private data structures. After **auth_destroy()** is called, *auth* is undefined.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **authnone_create()**, p. 77
- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
AUTH *authnone_create()
```

Library

rpcLib.a

Description

authnone_create() creates and returns a default RPC authentication handle that passes NULL authentication information with each remote procedure call.

If unsuccessful, **authnone_create()** returns a NULL.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **auth_destroy()**, **AUTH_DESTROY()**, p. 76
- **authunix_create()**, p. 78
- **authunix_create_default()**, p. 79
- **rpc_thread_init()**, p. 676
- **svcerr_auth()**, p. 828

authunix_create()

Synopsis

```
#include <rpc/rpc.h>
```

```
AUTH *authunix_create( char *host, int uid, int gid, int len, int *aupgids )
```

Library

rpcLib.a

Description

authunix_create() creates and returns an RPC authentication handle that has UNIX permissions.

host points to the name of the machine on which the permissions were created.

uid specifies the calling thread's effective user ID; *gid* specifies the calling thread's effective group ID.

len specifies the length of the groups array pointed to by *aupgids*.

If unsuccessful, **authunix_create()** returns a NULL.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **auth_destroy()**, **AUTH_DESTROY()**, p. 76
- **authnone_create()**, p. 77
- **authunix_create_default()**, p. 79
- **rpc_thread_init()**, p. 676
- **svcerr_auth()**, p. 828

Synopsis

```
#include <rpc/rpc.h>
AUTH *authunix_create_default()
```

Library

rpcLib.a

Description

authunix_create_default() calls **authunix_create()** to create a default UNIX RPC authentication handle.

If successful, **authunix_create_default()** returns the default RPC authentication handle. Otherwise, **authunix_create_default()** returns a NULL.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **auth_destroy()**, **AUTH_DESTROY()**, p. 76
- **authunix_create()**, p. 78
- **callrpc()**, p. 90
- **rpc_thread_init()**, p. 676
- **svcerr_auth()**, p. 828

bind()

Synopsis

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

int bind( int s, const struct sockaddr *name, int namelen );
```

Library

tcpipLib.a

Description

bind() assigns a name to a unnamed socket.

A socket *s* created using **socket()** exists in a name space (address family), but is unnamed. **bind()** assigns *name* to a socket; *name* points to an address structure, *sockaddr*, that specifies the address to which the socket is bound.

namelen specifies the length of *sockaddr*.

If successful, **bind()** returns 0. Otherwise, **bind()** returns -1 and sets *errno*.

Errors

[EBADF]	The descriptor <i>s</i> is not valid.
[EINVAL]	The socket is already bound to an address. <i>namelen</i> is greater than MLEN (MLEN is defined in file mbuf.h).
[ENOTSOCK]	The descriptor <i>s</i> references a file, not a socket.
[EADDRNOTAVAIL]	The specified address is unavailable from the local machine.
[EADDRINUSE]	The specified address is in use.

Example

The following example sets up a daemon socket.

```
int ex_bind(void)
{
    int          sd;
    struct sockaddr_in sa;
    struct servent *service;
    int          rc;

    bzero((char *)&sa, sizeof(sa));
    if (getservlock())
        return -1;
    service=getservbyname("ftp","tcp");
    if (NULL==service) {
        (void)printf("getservbyname() failed, using default\n");
        sa.sin_port=21;
    }
}
```



```
    } else {
        sa.sin_port=service->s_port;
    }
    getservunlock();
    if ((sd=socket(AF_INET,SOCK_STREAM,0))<0) {
        (void)printf("ex_bind: socket() failed\n");
        return(-1);
    }
    sa.sin_addr.s_addr=INADDR_ANY;
    sa.sin_family=AF_INET;
    if (bind(sd, (struct sockaddr *)&sa, sizeof(struct sockaddr_in))<0) {
        (void)printf("ex_bind: bind() failed\n");
        return(-1);
    }
    if (listen(sd,5)<0) {
        printf("ex_bind: listen() failed\n");
        return(-1);
    }
    return(sd);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.4
- **accept()**, p. 52
- **getsockname()**, p. 295
- **listen()**, p. 386
- **socket()**, p. 760

bootp_request()

Synopsis

```
#include <netLib.h>
int bootp_request(
    unsigned long    requestor_ip,
    char             *hwaddr,
    char             *toaddr,
    char             *buff,
    int              *recv_count );
```

Library

netLib.h

Description

bootp_request() sends bootp request packet, and attempts to receive a response from the bootp server.

requestor_ip specifies the IP address of the machine sending the bootp request. If this address is not known, specify 0x00000000.

hwaddr points to a 6-byte array that holds the hardware source address.

toaddr specifies the IP address of the target bootp server. If neither the address of the bootp server nor the address of the machine sending the bootp request is known, specify *toaddr* as 255.255.255.255. If the address and network mask of the machine sending the bootp request are known, but the address of the bootp server is not known, derive the address of the bootp server from the following logical expression: $(requestor_ip \& mask) | (\sim mask)$, and set *toaddr* to the resulting value.

buff points to the buffer that holds the bootp response. This buffer should contain at least 300 bytes.

recv_count points to an integer that specifies the number of bytes received from bootp server. *recv_count* is set only if **bootp_request** is successful.

If successful, **bootp_request** returns 0. Otherwise, **bootp_request** returns EINVAL if a system error was encountered, or returns ETIMEDOUT if the bootp server did not respond.

When an IP interface with broadcast capability is used, such as token-ring or Ethernet, but the caller's IP address is not known, ensure that the caller's IP address is set to 0.0.0.0 (using the **ifconfig()** function).

This function cannot be used with the ROM Monitor network device driver.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

Example

Network libraries must be initialized before this function can be called.

```
char hwaddr[6]={0x10, 0x00, 0xab, 0xcd, 0xef, 0x55};
int  rcv_count;
int rc;
char b[512];
char str1[50];
(void)strcpy(str1, "ent0 0.0.0.0");
if (ifconfig(str1)!=0) {
(void)printf("ifconfig error\n");
return(-1);
}
rc=bootp_request(0x00000000, hwaddr, "255.255.255.255", b,
&rcv_count);
(void)printf("return code = %d\n", rc);
if (rc==0) {
(void)printf("received bytes = %d\n", rcv_count);
(void)dbmem_disp(b, rcv_count, NULL);
}
```

References

- [tcpip_init\(\)](#)
- [net_init\(\)](#)

bprintf()

Synopsis

```
#include <kadtLib.h>

int bprintf( char **buffer, char *format, ... );
```

Library

kadtLib.a

Description

bprintf() prints a formatted string to a buffer, or calls a print function that was previously set by calling **bprintf_set()**.

If *buffer* is not NULL, it must point to a character pointer. On return from **bprintf()** the formatted string is placed at the location of the character pointer and the pointer is updated to point to the string terminator. This allows the buffer to be filled with multiple calls to **bprintf()**. Otherwise, the formatted string will either be printed to the standard output or printed using the function set by **bprintf_set()**.

format has the same form and function as the *format-string* parameter of **printf()**.

bprintf() returns the number of characters written.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf_set()**, p. 85
- **printf()**, p. 500

Synopsis

```
#include <kadtLib.h>
void bprintf_set( int (*prtfptr)(const char *, va_list) );
```

Library

kadtLib.a

Description

bprintf_set() sets the print function for **bprintf()**.

If **bprintf_set()** is not called, the default print function is **vprintf()**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **bprintf()**, p. 84

bsearch()

Synopsis

```
#include <stdlib.h>

void *bsearch( const void *key,
               const void *base,
               size_t num,
               size_t size,
               int (*compar)(const void *element1, const void *element2) );
```

Library

cLib.a

Description

bsearch() performs a binary search of an array of *num* elements. Each element contains *size* bytes.

The array must be sorted in ascending order. *base* is a pointer to the base of the array. *key* names the value being sought.

compare is a pointer to a programmer-supplied routine that compares two array elements and returns a value specifying their relationship. **bsearch()** calls the routine one or more times during the search, passing pointers to two array elements on each call. The routine must compare the elements and return one of the following values:

Value	Meaning
Less than 0	<i>element1</i> is less than <i>element2</i>
0	<i>element1</i> is identical to <i>element2</i>
Greater than 0	<i>element1</i> is greater than <i>element2</i>

bsearch() returns a pointer to *key* in the array to which *base* points. If two keys are identical, the element that *key* points to is unspecified. If **bsearch()** cannot find *key*, it returns a NULL.

Errors

None.

Example

The following example initializes an array of 10 integers to random numbers, sorts the array in ascending order, reads a number from the standard input, and searches the array for the number.

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

int comp(int *first, int *second)
```

```
{
    return(*first - *second);
}

main()
{
    int data[10], key, i;
    int *ptr;
    unsigned int seed = 15;

    /* Initialize array with random numbers */
    for(i = 0; i < 10; i++)
        data[i] = rand_r(&seed);

    /* Sort array in ascending order */
    qsort((void *)&data[0], 10, sizeof(int), comp);

    /* Get a number from stdin */
    printf("Enter a number:");
    scanf("%d", &key);

    /* Search the array for the number */
    ptr = (int *)bsearch((void *)&key, (void *)&data[0], 10,
        sizeof(int), comp);
    if(ptr == NULL)
        printf("Number %d, NOT found\n", key);
    else
        printf("Number %d, found\n", key);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.5.1
- **qsort()**, p. 616

_callxlc()

Synopsis

```
#include <cLib.h>
int _callxlc(int,...)
```

Library

csLib.a

Description

_callxlc() invokes functions in the OPENBIOS ROM Monitor code from an OS Open application.

The first parameter is the address of the function descriptor of the OPENBIOS function. The subsequent parameters are for the OPENBIOS function, and are shifted so that the second parameter to **_callxlc** is passed to the OPENBIOS function as the first, the third is passed as the second, and so on. These parameters must not be floating point values.

Errors

None.

Example

See the file **usr_samp.c** file in the samples directory of the OS Open platform since this function must be called differently between XCOFF and ELF applications.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <stdlib.h>

void *calloc( size_t num, size_t size );
```

Library

cLib.a

Description

calloc() allocates memory for an array of *num* elements, each containing *size* bytes, and initializes each element to 0.

calloc() returns a pointer to the allocated memory, which is guaranteed to be suitably aligned for any type of object. To get a pointer to a type, use a type cast on the returned value.

If not enough memory is available, or if either *num* or *size* is 0, **calloc()** returns a NULL.

Errors

None.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.3.1
- **free()**, p. 228
- **malloc()**, p. 398
- **realloc()**, p. 638

callrpc()

Synopsis

```
#include <rpc/rpc.h>
#include <types.h>

enum clnt_stat callrpc( char *host, u_long prognum, u_long versnum,
                       u_long procnum, xdrproc_t inproc, char *in,
                       xdrproc_t outproc, char *out );
```

Library

rpcLib.a

Description

callrpc() calls the remote procedure identified by the *prognum*, *versnum* and *procnum* parameters on the machine pointed to by *host*.

callrpc() uses the User Datagram Protocol (UDP) as a transport to call a remote procedure. No connection is made if the server is supported by TCP/IP. **callrpc()** does not control timeouts or authentication.

prognum specifies the number of the remote program. *versnum* specifies the version number of the remote program. *procnum* specifies the number of the procedure associated with the remote program being called.

inproc names the external data representation (XDR) procedure that encodes the procedure parameters; *outproc* names the XDR procedure that decodes the procedure results. *in* specifies the address of the procedure arguments; *out* specifies the address where results are stored.

callrpc() returns a value of **enum clnt_stat**. **clnt_perrno()** can translate the failure value into a displayed message.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **clnt_broadcast()**, p. 101
- **clnttcp_create()**, p. 117
- **clntudp_create()**, p. 121
- **clnt_perrno()**, p. 111
- **registerrpc()**, p. 649
- **rpc_thread_init()**, p. 676
- **svc_run()**, p. 825

Synopsis

```
#include <math.h>
double ceil( double x );
```

Library

mathLib.a

Description

ceil() computes the smallest integral value not less than *x*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.6.1

cfgetispeed()

Synopsis

```
#include <termios.h>
speed_t cfgetispeed( const struct termios termios_p );
```

Library

ttyLib.a

Description

cfgetispeed() returns the input baud rate stored in the structure **termios** to which *termios_p* points.

The type **speed_t** is an unsigned integer type defined in the file **<termios.h>**.

cfgetispeed() returns the value found in *termios* without interpretation.

Errors

None.

Example

See **cfsetospeed()**, p. 95.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, § 7.1.3
- **cfgetospeed()**, p. 93
- **cfsetispeed()**, p. 94
- **cfsetospeed()**, p. 95
- **tcsetattr()**, p. 860

Synopsis

```
#include <termios.h>

speed_t cfgetospeed( const struct termios termios_p );
```

Library

ttyLib.a

Description

cfgetospeed() returns the output baud rate stored in the structure **termios** pointed to by *termios_p*.

The type **speed_t** is an unsigned integer type defined in the file **<termios.h>**.

cfgetospeed() returns the value found in *termios* without interpretation.

Errors

None.

Example

See **cfsetospeed()**, p. 95.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, § 7.1.3
- **cfgetispeed()**, p. 92
- **cfsetispeed()**, p. 94
- **cfsetospeed()**, p. 95
- **tcsetattr()**, p. 860

cfsetispeed()

Synopsis

```
#include <termios.h>

int cfsetispeed( struct termios termios_p, speed_t speed );
```

Library

ttyLib.a

Description

cfsetispeed() sets the input baud rate, stored in the structure **termios** pointed to by *termios_p*, to *speed*.

The specified terminal device remains unaffected until **tcsetattr()** is called. Certain values for speeds passed to **tcsetattr()** have special meanings, which are discussed in **tcsetattr()** on page 860.

The type **speed_t** is an unsigned integer type defined in the file **<termios.h>**.

If successful, **cfsetispeed()** returns 0, but **cfsetispeed()** does not check whether the input baud rate is supported. Otherwise, **cfsetispeed()** returns -1.

Errors

None.

Example

See **cfsetospeed()**, p. 95.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, § 7.1.3
- **cfgetispeed()**, p. 92
- **cfgetospeed()**, p. 93
- **cfsetospeed()**, p. 95
- **tcsetattr()**, p. 860

Synopsis

```
#include <termios.h>

int cfsetospeed( struct termios termios_p, speed_t speed );
```

Library

ttyLib.a

Description

cfsetospeed() sets the output baud rate, stored in the **termios** structure pointed to by *termios_p*, to *speed*.

The effects on the terminal device do not occur until **tcsetattr()** is called. Certain values for speeds that are passed to **tcsetattr()** have special meanings. These values are discussed in **tcsetattr()** on page 860.

The type **speed_t** is an unsigned integer type defined in the file **<termios.h>**.

If successful, **cfsetospeed()** returns 0, but **cfsettospeed()** does not check whether the output baud rate is supported. Otherwise, **cfsettospeed()** returns -1.

Errors

None.

Example

The following example shows a function that gets the current input and output baud rates, saves their values, and sets the input and output speeds to 38400 baud.

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

speed_t ispeed, ospeed;

void ttySpeed() {
    int rc;
    struct termios ttyAttr;

    rc = tcgetattr(STDIN_FILENO, &ttyAttr);
    if (rc == 0) {
        ispeed = cfgetispeed(&ttyAttr);    /* Save input speed */
        ospeed = cfgetospeed(&ttyAttr);    /* Save output speed */
        cfsetospeed(&ttyAttr, B38400);    /* Set input speed */
    }
}
```

cfsetospeed()

```
        cfsetospeed(&ttyAttr, B38400);    /* Set output speed */
        /* Set new speeds after current input and output are complete */
        tcsetattr(STDIN_FILENO, TCSAFLUSH, &ttyAttr);
    }
} /* end ttySpeed() */
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, § 7.1.3
- **cfgetispeed()**, p. 92
- **cfgetospeed()**, p. 93
- **cfsetispeed()**, p. 94
- **tcsetattr()**, p. 860

Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod( const char *path, mode_t mode );
```

Library

devLib.a

Description

chmod() changes the file permission bits of the file pointed to by *path* to the values specified in *mode*.

If successful, **chmod()** returns 0. Otherwise, **chmod()** returns -1 and sets *errno*.

Errors

[EACCES]	Search permission is denied for a component of the path.
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of <i>path</i> exceeds {PATH_MAX} .
[ENOENT]	<i>path</i> names a file that does not exist or points to an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The file permissions do not allow the mode to be altered.
[EROFS]	<i>path</i> resides on a read-only file system

Example

See **mkdir()**, p. 427.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.6.4

clearDisp()

Synopsis

```
#include <shell.h>
void clearDisp();
```

Library

shell.o

Description

clearDisp() writes characters to the standard output that clear the screen.

Note: **clearDisp()** is for DEC VT100 terminals and emulations only.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

Synopsis

```
#include <stdio.h>

void clearerr( FILE *stream );
```

Library

fsLib.a

Description

clearerr() resets the error indicator and end-of-file indicator for *stream*.

The indicators remain set until the calling program calls **clearerr()** or **rewind()**. The **fseek()** function also clears the end-of-file (EOF) indicator.

Note: The file system must have been initialized by **fs_init**.

Errors

None.

Example

See **puts()**, p. 615.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.10.1
- **feof()**, p. 195
- **ferror()**, p. 196
- **fs_init()**, p. 231
- **fseek()**, p. 235
- **perror()**, p. 488
- **rewind()**, p. 652

cLib_init()

Synopsis

```
#include <cLib.h>
int cLib_init(void);
```

Library

cLib.a

Description

cLib_init() initializes the ANSI C Library runtime environment for **OS OPEN**.

cLib_init() must be called from the kernel thread group before **malloc()**, **free()**, **calloc()**, **realloc()**, **signal()**, or **raise()** can be called.

If unsuccessful, **cLib_init()** returns -1, Otherwise, **cLib_init()** returns 0.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	No

References

- **malloc()**, p. 398
- **signal()**, p. 746
- **raise()**, p. 630
- **free()**, p. 228
- **calloc()**, p. 89
- **realloc()**, p. 638

Synopsis

```
#include <rpc/rpc.h>

enum clnt_stat clnt_broadcast( char *host, u_long prognum,
                               u_long versnum, u_long procnum,
                               xdrproc_t inproc, char *in,
                               xdrproc_t outproc, char *out,
                               resultproc_t eachresult );
```

Library

rpcLib.a

Description

clnt_broadcast() broadcasts a remote procedure call to all hosts on a network.

prognum specifies the number of the remote program. *versnum* specifies the version number of the remote program. *procnum* specifies the number of the procedure associated with the remote program being called.

inproc names the external data representation (XDR) procedure that encodes the procedure parameters. *outproc* names the XDR procedure that decodes the procedure results. *in* specifies the address of the procedure arguments; *out* specifies the address where results are stored.

When a client broadcasts a remote procedure call, multiple server programs may respond. When the client receives a response, **clnt_broadcast()** calls an *eachresult* function of the form:

```
eachresult(out, *addr)
char *out;
struct sockaddr_in *addr;
```

addr specifies the address of the server that sent the result.

Broadcast sockets are limited in size to the maximum transfer unit of the data link.

If the *eachresult* function returns 0, **clnt_broadcast()** waits for more replies. Otherwise, **clnt_broadcast()** returns a value of **enum clnt_stat**, which **clnt_perrno()** can translate into a displayed message.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

clnt_broadcast()

References

- `callrpc()`, p. 90
- `rpc_thread_init()`, p. 676

Synopsis

```
#include <rpc/rpc.h>
#include <types.h>

enum clnt_stat callrpc( char *host, u_long prognum, u_long versnum,
                      u_long procnum, xdrproc_t inproc, char *in,
                      xdrproc_t outproc, char *out, struct timeval tout );
```

Library

None. This is a macro defined in `<rpc/clnt.h>`.

Description

clnt_call(), also called **CLNT_CALL()**, calls the remote procedure associated with the client handle structure pointed to by *clnt*.

The client handle structure pointed to by *clnt* results from an RPC client creation function, such as **clntudp_create()**, which opens a User Datagram Protocol (UDP) socket.

prognum specifies the number of the remote program. *versnum* specifies the version number of the remote program. *procnum* specifies the number of the procedure associated with the remote program being called.

inproc names the XDR procedure that encodes the procedure parameters. *outproc* names the XDR procedure that decodes the procedure results. *in* specifies the address of the procedure arguments; *out* specifies the address where results are stored.

tout specifies the time allowed for results to return.

clnt_call() returns a value of **enum clnt_stat**, which **clnt_perrno()** can translate into a displayed message.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **callrpc()**, p. 90
- **clnt_perror()**, p. 112
- **clnttcp_create()**, p. 117
- **clntudp_create()**, p. 121
- **rpc_thread_init()**, p. 676

clnt_control(), CLNT_CONTROL()

Synopsis

```
#include <rpc/rpc.h>
bool_t clnt_control( CLIENT *cl, int req, char *info )
```

Library

None. This is a macro defined in `<rpc/clnt.h>`.

Description

clnt_control(), also called **CLNT_CONTROL()**, changes or retrieves information about a client object.

cl points to the client handle structure that results from an RPC client creation subroutine, such as **clntudp_create()**.

UDP and TCP support the following values for *req*, which indicates the request type:

CLSET_TIMEOUT Sets the total timeout value (**struct timeval**)

CLGET_TIMEOUT Gets the total timeout value (**struct timeval**)

If a timeout is set using **clnt_control()**, the timeout value passed to **clnt_call()** is ignored in future calls.

CLGET_SERVER_ADDR

Gets the address of the server (**struct sockaddr**)

CLSET_RETRY_TIMEOUT

UDP only; sets the retry timeout value (**struct timeval**)

CLGET_RETRY_TIMEOUT

UDP only; gets the retry timeout value (**struct timeval**)

The retry timeout value is the time UDP/RPC waits for the server to reply before retransmitting.

info points to the structure associated with the request type.

If successful, **clnt_control()** returns 1. Otherwise, **clnt_control()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **clnt_call(), CLNT_CALL()**, p. 103
- **clnt_create()**, p. 106
- **clntudp_create()**, p. 121
- **rpc_thread_init()**, p. 676

clnt_create()

Synopsis

```
#include <rpc/rpc.h>

CLIENT *clnt_create( char *host, unsigned prognum, unsigned versnum,
                    char *protocol );
```

Library

rpcLib.a

Description

clnt_create() creates and returns a generic client handle.

host names the remote host where the server is located.

prognum specifies the number of the remote program; *version* specifies its version number.

protocol specifies the transport protocol used by the calling program, either UDP or TCP.

RPC messages transported by UDP can contain up to 8KB of encoded data. Use this transport for procedures that handle less than 8KB of argument or result data.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **clnt_control()**, **CLNT_CONTROL()**, p. 104
- **clnt_destroy()**, **CLNT_DESTROY()**, p. 107
- **clnttcp_create()**, p. 117
- **clntudp_create()**, p. 121
- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>

void clnt_destroy( CLIENT *clnt );
```

Library

None. This is a macro defined in `<rpc/clnt.h>`.

Description

clnt_destroy(), also called **CLNT_DESTROY()**, destroys the client handle structure pointed to by *clnt*.

Destroying the client handle deallocates private data structures, including the object pointed to by *clnt*, which becomes undefined after **clnt_destroy()** is called. The client socket, if open, is also closed.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **clnttcp_create()**, p. 117
- **clntudp_create()**, p. 121
- **rpc_thread_init()**, p. 676

clnt_freeres(), CLNT_FREERES()

Synopsis

#include <rpc/rpc.h>

bool_t clnt_freeres(CLIENT *clnt, xdrproc_t outproc, char *out)

Library

None. This is a macro defined in **<rpc/clnt.h>**.

Description

clnt_freeres(), also called **CLNT_FREERES()**, frees data previously allocated by RPC/XDR.

The data was allocated when RPC/XDR decoded the results of a remote procedure call.

clnt points to the client handle structure.

outproc specifies an XDR function that describes the results in simple decoding primitives, such as **xdr_bytes()**, **xdr_char()**, and so on.

out points to the address of the stored results.

clnt_freeres() returns the value of *outproc*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>

void clnt_geterr( CLIENT *clnt, struct rpc_err *errp );
```

Library

None. This is a macro defined in `<rpc/clnt.h>`.

Description

clnt_geterr(), also called **CLNT_GETERR()**, copies error information from the client RPC handle pointed to by *clnt* to an error structure pointed to by *errp*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **clnt_pcreateerror()**, p. 110
- **clnt_perrno()**, p. 111
- **clnt_perror()**, p. 112
- **clnt_spccreateerror()**, p. 113
- **clnt_sperrno()**, p. 114
- **clnt_sperror()**, p. 115
- **rpc_thread_init()**, p. 676

clnt_pcreateerror()

Synopsis

```
#include <rpc/rpc.h>
void clnt_pcreateerror( char *s );
```

Library

rpcLib.a

Description

clnt_pcreateerror() writes a message to the standard error describing why a client RPC handle could not be created.

The message is preceded by the string pointed by *s* and a colon.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **clnt_create()**, p. 106
- **clnt_geterr()**, **CLNT_GETERR()**, p. 109
- **clntraw_create()**, p. 116
- **clnttcp_create()**, p. 117
- **clntudp_create()**, p. 121
- **clnt_spccreateerror()**, p. 113
- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
void clnt_perrno( enum clnt_stat stat );
```

Library

rpcLib.a

Description

clnt_perrno() writes a message to the standard error corresponding to the condition named by *stat*, which specifies the client error status of the remote procedure call.

clnt_perrno() is used when **callrpc()** fails. **clnt_perrno()** translates the failure status named by *stat* into a message.

If the program does not write to the standard error, or does not write the message using **fprintf()**, **clnt_sperrno()** should be used instead.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **callrpc()**, p. 90
- **clnt_geterr()**, **CLNT_GETERR()**, p. 109
- **clnt_pcreateerror()**, p. 110
- **clnt_sperrno()**, p. 114
- **rpc_thread_init()**, p. 676

clnt_perror()

Synopsis

```
#include <rpc/rpc.h>
void clnt_perror( CLIENT *clnt, char *s );
```

Library

rpcLib.a

Description

clnt_perror() writes a message to the standard error indicating why a remote procedure call failed.

The message is prepended with the string pointed to by *s* followed by a colon.

clnt points to the client handle structure used in the remote procedure call.

clnt_perror() can be used after **clnt_call()**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **clnt_call()**, **CLNT_CALL()**, p. 103
- **clnt_geterr()**, **CLNT_GETERR()**, p. 109
- **clnt_pcreateerror()**, p. 110
- **clnt_sperror()**, p. 115
- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
char *clnt_spcreateerror( char *s );
```

Library

rpcLib.a

Description

clnt_spcreateerror() returns a pointer to an error message preceded by the string pointed to by *s*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **clnt_geterr()**, **CLNT_GETERR()**, p. 109
- **clnt_pcreateerror()**, p. 110
- **rpc_thread_init()**, p. 676

clnt_sperrno()

Synopsis

```
#include <rpc/rpc.h>
char *clnt_sperrno( enum clnt_stat stat );
```

Library

rpcLib.a

Description

clnt_sperrno() returns a pointer to a string containing a status message specified by *stat*.

The status message specifies the client error status of the remote procedure call and ends with a newline character. The status message is one of a set of messages defined in the file **<rpc/clnt.h>**.

clnt_sperror() can translate and display the returned status message.

If one of the following conditions exists when **callrpc()** fails, **clnt_sperrno()** is used instead of **clnt_perrno()**.

- The program does not write to the standard error
- The status message is not output using **printf()**
- A message format not supported by **clnt_perrno()** is used

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **clnt_geterr()**, **CLNT_GETERR()**, p. 109
- **clnt_perrno()**, p. 111
- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
char *clnt_sperror( CLIENT *cl; char *s );
```

Library

rpcLib.a

Description

clnt_sperror() writes the string pointed to by *s* to the standard error.

The string contains a status message indicating why a remote procedure call failed. The message is obtained using **clnt_sperrno()**.

cl points to a client handle structure.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **clnt_geterr()**, **CLNT_GETERR()**, p. 109
- **clnt_perror()**, p. 112
- **clnt_sperrno()**, p. 114
- **rpc_thread_init()**, p. 676

clntraw_create()

Synopsis

```
#include <rpc/rpc.h>
```

```
CLIENT *clntraw_create( u_long prognum, u_long versnum );
```

Library

rpcLib.a

Description

clntraw_create() creates an RPC client used to simulate a remote program.

prognum specifies the program number of the remote program; *versnum* specifies its version number.

The RPC client uses a buffer private to the calling thread.

If successful, **clntraw_create()** returns a pointer to a valid RPC client. Otherwise, **clntraw_create()** returns a NULL.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **clnt_pcreateerror()**, p. 110
- **rpc_thread_init()**, p. 676
- **svcrw_create()**, p. 836

Synopsis

```
#include <types.h>
#include <netinet/in.h>
#include <rpc/rpc.h>

CLIENT *clnttcp_create( struct sockaddr_in *addr, u_long prognum,
                        u_long versnum, int *sockp, u_int sendsz,
                        u_int recvsz );
```

Library

rpcLib.a

Description

clnttcp_create() creates an RPC client transport handle for a remote program.

The client uses TCP as the transport to the service.

addr points to the Internet address of the remote program. If the **sin_port** field in **struct sockaddr_in** associated with the Internet address is 0, *addr* is set to the port on which the remote program is listening. The remote procedure call consults the remote portmap daemon to obtain the port information.

prognum specifies the program number of the remote program; *versnum* specifies its version number.

sockp points to a socket. If the value of *sockp* is **RPC_ANYSOCK**, **clnttcp_create()** opens a new socket and binds *sockp* to the port number obtained previously.

TCP/IP remote procedure calls use buffered I/O. *sendsz* and *recvsz* specify the size of the send and receive buffers. If either parameter is set to 0, **clnttcp_create()** assigns default sizes.

If successful, **clnttcp_create()** returns a valid TCP/IP client handle. Otherwise, **clnttcp_create()** returns a value of NULL.

Errors

[EADDINUSE]	The port is in use
[EPFNOSUPPORT]	The protocol family is not supported.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

clnttcp_create()

References

- `callrpc()`, p. 90
- `clnt_call()`, `CLNT_CALL()`, p. 103
- `clnt_pcreateerror()`, p. 110
- `clntudp_create()`, p. 121
- `rpc_thread_init()`, p. 676
- `svctcp_create()`, p. 837

Synopsis

```
#include <types.h>
#include <netinet/in.h>
#include <rpc/clnt.h>
#include <rpc/rpc.h>

CLIENT *clntudp_bufcreate( struct sockaddr_in *addr, u_long program,
                           u_long version, struct timeval wait, int *sockp,
                           u_int sendsz, u_int recvz );
```

Library

rpcLib.a

Description

clntudp_bufcreate() creates an RPC client transport handle for a remote program.

The client specifies send and receive buffer sizes and uses UDP as the transport to the service.

addr points to the IP address of the remote program. If the **sin_port** field in **struct sockaddr_in** associated with the IP address is 0, *addr* is set to the port on which the remote program listens. The client making the remote procedure call consults the remote portmap daemon to obtain port information.

program specifies the program number of the remote program; *versnum* specifies its version number.

sockp points to a socket. If the value of *sockp* is **RPC_ANYSOCK** (defined in the file **<rpc.svc.h>**), **clntudp_bufcreate()** opens a new socket and binds *sockp* to the port.

UDP resends the request in intervals specified in the *wait* structure.

sendsz specifies the buffer size for outgoing data; *recvsz* specifies the buffer size for incoming data.

If successful, **clntudp_bufcreate()** returns a valid RPC client transport handle. Otherwise, **clntudp_bufcreate()** returns a NULL.

Errors

[EADDINUSE]	The port is already in use
[EPFNOSUPPORT]	The protocol family is not supported.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

clntudp_bufcreate()

References

- `clntraw_create()`, p. 116
- `clnttcp_create()`, p. 117
- `clntudp_create()`, p. 121
- `rpc_thread_init()`, p. 676

Synopsis

```
#include <types.h>
#include <netinet/in.h>
#include <rpc/rpc.h>

CLIENT *clntudp_create( struct sockaddr_in *addr, u_long prognum,
                        u_long versnum, struct timeval wait, int *sockp );
```

Library

rpcLib.a

Description

clntudp_create() creates an RPC client transport handle for a remote program.

The client uses UDP as the transport to the service.

RPC messages transported by UDP can contain up to 8KB of encoded data; **clntudp_create()** is appropriate for programs that take arguments or return results smaller than 8KB.

addr points to the IP address of the remote program. If the **sin_port** field in **struct sockaddr_in** associated with the IP address is 0, *addr* is set to the port on which the remote program listens. The client making the remote procedure call consults the remote portmap daemon to obtain port information.

prognum specifies the program number of the remote program; *versnum* specifies its version number.

wait sets the amount of time the UDP transport waits for a response before the transport sends another remote procedure call, or until the remote procedure call times out. The total time for the remote procedure call to time out is set using **clnt_call()**.

sockp specifies a pointer to a socket. If the value of *sockp* is **RPC_ANYSOCK** (defined in the file **<rpc.svc.h>**), **clntudp_create()** opens a new socket and sets *sockp* to the port.

If successful, **clntudp_create()** returns a valid UDP/IP client handle. Otherwise, **clntudp_create()** returns a NULL.

Errors

[EADDINUSE]	The port is already in use
[EPFNOSUPPORT]	The protocol family is not supported.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

clntudp_create()

References

- `callrpc()`, p. 90
- `clnt_call()`, `CLNT_CALL()`, p. 103
- `clnt_pcreateerror()`, p. 110
- `clntudp_create()`, p. 121
- `rpc_thread_init()`, p. 676
- `svcudp_create()`, p. 839

Synopsis

```
#include <time.h>
clock_t clock ( void );
```

Library

None. This is a macro defined in **<time.h>**.

Description

OS Open supports one system-wide real-time clock, **CLOCK_REALTIME**, defined in the file **<time.h>**.

clock() returns -1 .

clock() is provided for ANSI C compatibility.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.12.2.1

clock_getres()

Synopsis

```
#include <time.h>

int clock_getres( clockid_t clock_id, timespec_t *res );
```

Library

rtxLib.a

Description

clock_getres() gets the resolution in nanoseconds of the clock named by *clock_id*.

OS Open supports one system-wide real-time clock, **CLOCK_REALTIME**, defined in the file **<time.h>**.

If successful, **clock_getres()** returns 0 and stores the resolution of the named clock in the field pointed to by *res*. Otherwise, **clock_getres()** returns -1 and sets *errno*; the field pointed to by *res* does not change.

Errors

[EINVAL] *clock_id* does not name a known clock.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §14.2.1
- **clock_gettime()**, p. 125
- **clock_settime()**, p. 126

Synopsis

```
#include <time.h>

int clock_gettime( clockid_t clock_id, timespec_t *tp );
```

Library

rtxLib.a

Description

clock_gettime() gets the current time from the clock named by *clock_id* and stores the current time in the field pointed to by *tp*.

OS Open supports one system-wide real-time clock, **CLOCK_REALTIME**, defined in the file **<time.h>**.

The current time is specified as the elapsed time in seconds and nanoseconds since 00:00:00 UTC, January 1, 1970.

If successful, **clock_gettime()** returns 0 and updates the field pointed to by *tp*. Otherwise, **clock_gettime()** returns -1 and sets *errno*. The field pointed to by *tp* does not change.

If the real-time clock has never been set using **clock_settime()**, the result of a call to **clock_gettime()** is undefined.

Errors

[EINVAL] *clock_id* does not name a known clock.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §14.2.1
- **clock_getres()**, p. 124
- **clock_settime()**, p. 126

clock_settime()

Synopsis

```
#include <time.h>

int clock_settime( clockid_t clock_id, timespec_t *tp );
```

Library

rtxLib.a

Description

clock_settime() sets the clock named by *clock_id*.

OS Open supports one system-wide real-time clock, **CLOCK_REALTIME**, defined in the file **<time.h>**.

The field pointed to by *tp* should contain the current time, which is the elapsed time in seconds and nanoseconds since the epoch (00:00:00 UTC, January 1, 1970).

If successful, **clock_settime()** returns 0. Otherwise, **clock_settime()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>clock_id</i> does not specify a known clock, or the <i>tp</i> nanosecond field is either greater than 1000 million or less than 0.
----------	---

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §14.2.1
- **clock_getres()**, p. 124
- **clock_gettime()**, p. 125

Synopsis

```
#include <unistd.h>
int close( int filedes );
```

Library

devLib.a

Description

close() deallocates the file descriptor named by *filedes*.

For logical file systems (type **LFSTYPE**), the driver-specific **close()** is called whenever a file descriptor is closed for that file system.

For devices, the call to the driver-specific **close()** is deferred until all open files on the device are closed.

If successful, **close()** returns 0. Otherwise, **close()** returns -1 and sets *errno*.

Errors

[EBADF] *filedes* does not name a valid file descriptor.

Example

See **read()**, p. 633.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §6.3.1
- **open()**, p. 478
- **opendir()**, p. 480

closedir()

Synopsis

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

int closedir( DIR *dp );
```

Library

devLib.a

Description

closedir() closes the directory stream pointed to by *dp*.

If successful, **closedir()** returns 0. Otherwise, **closedir()** returns -1 and sets *errno*.

Errors

[EBADF] *dp* does not point to an open directory stream.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.1.2.2
- **opendir()**, p. 480

Synopsis

```
#include <kadtLib.h>

int cond_dump( pthread_cond_t condid, char *buf );
```

Library

kadtLib.a

Description

cond_dump() provides information about the conditional variable named by *condid*.

If *buf* is NULL, the output is written to the standard output. Otherwise the output is stored in the buffer pointed to by *buf*.

If *condid* is not a valid condition, **cond_dump()** returns -1. Otherwise, **cond_dump()** returns 0.

Errors

None.

Example

The following example shows the information about conditional variable ID 0x41b4de.

```
OS OPEN>cond_dump(0x41b4de)
```

```
-----
Cond ID: 0x41b4de
```

```
Associated mutex ID: none
```

```
Waiting threads:
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

cond_list()

Synopsis

```
#include <kadtLib.h>
int cond_list( char **bufp );
```

Library

kadtLib.a

Description

cond_list() provides formatted information about all condition variables.

The information is part of the **kda_dump()** display.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

cond_list() returns 0.

Errors

None.

Example

The following example shows a listing of all condition variables.

```
OS OPEN>cond_list()
```

```
Condition
```

```
0x41b4de
0x41b462
0x41b494
0x41b418
0x41b3e0
0x425c22
0x425ba6
0x425bd8
0x425b5c
0x425b24
0x460c12
0x460b96
0x460bc8
0x460b4c
0x460b14
0x27af9c
0x27aec0
0x4bd828
0x4bdc38
```

0x158548

----- Total Number of Conditions: 20 -----

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **kda_dump()**, p. 349

connect()

Synopsis

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

int connect( int s, const struct sockaddr *name, int namelen );
```

Library

tcpipLib.a

Description

connect() connects two sockets.

The argument *s* names a socket. If *s* is of the type **SOCK_DGRAM**, **connect()** names the peer with which *s* is associated. The address of the peer is where datagrams are sent, and the only address from which datagrams are received.

If the socket is of type **SOCK_STREAM**, **connect()** tries to connect to another socket. *name* specifies the other socket, an address in the communication space of the local socket. Every communication space interprets *name* in its own way. Generally, stream sockets can connect only once.

Datagram sockets can call **connect()** many times to change their association. They can connect to an address that is no valid, such as a null address, to dissolve the association. If **connect()** is issued on an unbound socket, OS Open automatically binds the socket.

If successful, **connect()** returns 0. Otherwise, **connect()** returns -1 and sets *errno*.

Errors

[EBADF]	The descriptor <i>s</i> is not valid.
[ENOTSOCK]	The descriptor <i>s</i> references a file, not a socket.
[EADDRNOTAVAIL]	The named address is unavailable from the local machine.
[EAFNOTSUPPORT]	An address in the named address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network is not reachable from the host.
[EADDRINUSE]	The named address is busy.
[EINPROGRESS]	The socket is non-blocking and the connection cannot be completed immediately.
[EALREADY]	The socket is non-blocking and a previous connection attempt has not finished.

Example

The following example creates a socket and connects it to an address.

```
int          s;
struct sockaddr_in sa;
            :
            :
sa.sin_addr.s_addr=0x09437D03;
sa.sin_family=AF_INET;
sa.sin_port=htons(21); /* port number for ftp */
s=socket(AF_INET, SOCK_STREAM, 0)
if (s<0) {
    (void)printf("socket() call failed\n");
    return(-1);
}
if (connect(s, (struct sockaddr *)&sa sizeof(struct sockaddr)) < 0) {
    (void)printf("connect() call failed\n");
    return(-1);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.5
- **accept()**, p. 52
- **getsockname()**, p. 295
- **socket()**, p. 760

cos()

Synopsis

```
#include <math.h>
double cos( double x );
```

Library

mathLib.a

Description

cos() computes the cosine of x (measured in radians). The returned cosine is in the range $[-1, +1]$ for all real x .

Errors

[ERANGE]	The result is out of range.
----------	-----------------------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.2.5

Synopsis

```
#include <math.h>
double cosh( double x );
```

Library

mathLib.a

Description

cosh() computes the hyperbolic cosine of *x*.

If the magnitude of *x* is too large, a range error occurs and **cosh()** sets *errno*.

Errors

[ERANGE]	The magnitude of <i>x</i> is too large.
----------	---

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.3.1

cpp_exit()

Synopsis

```
#include <cppprt.h>
void cpp_exit( void )
```

Library

cppprtLib.a (XCOFF only)
mwdctor.o (ELF only, base object)
mwdctorl.o (ELF only, dynamically loadable objects)

Description

cpp_exit() removes the C++ runtime environment initialized by **cpp_init()** and calls the static destructors defined in the load image. Note that while this function is async, cancel, and interrupt safe, functions used in static destructors could change any or all of these attributes.

There are no errors.

Errors

None

Example

The following shows the calling syntax:

```
// This function should be called after use of the C++ environment
// is complete
cpp_exit();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **cpp_init()**, p. 137

Synopsis

```
#include <cpprt.h>
void cpp_init( void )
```

Library

cpprtLib.a (XCOFF only)
mwdctor.o (ELF only, base object)
mwdctorl.o (ELF only, dynamically loadable objects)

Description

cpp_init() initializes the C++ runtime environment and calls static constructors defined in the load image. Note that while this function is async, cancel, and interrupt safe, functions used in static constructors could change any or all of these attributes.

There are no errors.

Errors

None

Example

The following shows the calling syntax:

```
// This function must be called before any methods utilizing instances
// initialized via static constructors
cpp_init();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **cpp_exit()**, p. 136

cs_card_status()

Synopsis

```
#include <sys/csLib.h>
int cs_card_status( int card_id );
```

Library

csLib.a

Description

cs_card_status() returns information about the insertion state of the PC Card identified by *card_id* and stores the insertion state information.

cs_card_status() returns 0 if the specified PC Card is plugged into the system and was inserted before the most recent call to **cs_card_status()**.

cs_card_status() returns 1 if the card is plugged in, but was inserted after the most recent call to **cs_card_status()**. If the specified PC Card is not plugged into the system, or if *card_id* is invalid, **cs_card_status()** returns -1.

Before **cs_card_status()** is called, **cs_init()** must be called to initialize the Card Services/Enabler software layer.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **cs_init()**, p. 141
- **cs_stateless_card_status()**, p. 148
- Chapter 11, "OS Open PCMCIA Support," in *OS Open User's Guide*

Synopsis

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

int cs_get_io_window( int Adapter, int Socket );
```

Library

csLib.a

Description

cs_get_io_window() reserves an available I/O window for the socket, specified by *Socket* on the adapter specified by *Adapter*, and returns the window number. The window remains reserved until the PC Card plugged into the specified socket is removed.

When the PC Card is removed, the I/O window is disabled and made available for a subsequent **cs_get_io_window()** call.

Before **cs_card_status()** is called, **cs_init()** must be called to initialize the PCMCIA Card Services/Enabler software layer.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **cs_get_mem_window()**, p. 140
- **cs_init()**, p. 141
- Chapter 11, "OS Open PCMCIA Support," in *OS Open User's Guide*

cs_get_mem_window()

Synopsis

```
#include <sys/csLib.h>
int cs_get_mem_window( int Adapter, int Socket );
```

Library

csLib.a

Description

cs_get_mem_window() reserves an available memory window for the socket, specified by *Socket* on the adapter specified by *Adapter*, and returns the window number. The window remains reserved until the PC Card plugged into the specified socket is removed.

When the PC Card is removed, the memory window is disabled and made available for a subsequent **cs_get_mem_window()** call.

Before **cs_card_status()** is called, **cs_init()** must be called to initialize the PCMCIA Card Services/Enabler software layer.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **cs_get_io_window()**, p. 139
- **cs_init()**, p. 141
- Chapter 11, "OS Open PCMCIA Support," in OS Open *User's Guide*

Synopsis

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

int cs_init( event_t sc_event, IRQ sc_routing, void *ISA_mem_start,
            void *first_cs_address, int cards, ... );
```

Library

csLib.a

Description

cs_init() initializes the PCMCIA Card Services/Enabler (CS/Enabler) software layer.

sc_event specifies the OS Open event for socket status change interrupts;
sc_routing specifies the ISA interrupt routing (number) for the event.

ISA_mem_start specifies the system address that corresponds to ISA address 0.

cards specifies the number of PC Cards to be handled by the CS/Enabler software layer. Following *cards*, a pointer to a **cs_card_info_struct** structure must be passed to **cs_init()** for each PC Card represented by *cards*.

The CS/Enabler software layer reserves 4096-byte blocks of ISA memory space for accessing the Card Information Structures and configuration registers of PC Cards. One block is reserved for each socket. If the Card Information Structure and configuration registers of a PC Card represented by *cards* do not fit in 4096 bytes, an additional block is reserved for that PC Card. *first_address* specifies the starting system address of the first (highest) reserved block. Additional required blocks are reserved from system memory immediately below the most recently reserved block.

Before **cs_init()** is called, **ss_init()** must be called to initialize PCMCIA Socket Services.

If successful, **cs_init()** returns 0, indicating that the CS/Enabler software layer is initialized and using PCMCIA Socket Services to control the PCMCIA sockets.

Errors

[EINVAL]	Invalid members in the passed cs_card_info_struct structure.
----------	---

Example

See “Putting it All Together” on page 11-25 in *OS Open User's Guide*.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

cs_init()

References

- **ss_init()**, p. 766
- Chapter 11, “OS Open PCMCIA Support,” in *OS Open User's Guide*

Synopsis

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

int cs_int_install( int card_id, int event, flih_t *new_flih, flih_t *old_flih );
```

Library

csLib.a

Description

cs_int_install() installs a first-level interrupt handler (FLIH) for the event named by *event* for the PC Card identified by *card_id*. **cs_int_install()** is used instead of **ext_int_install()** when multiple PC Cards share an OS Open event.

The *event*, *new_flih*, and *old_flih* parameters for **cs_int_install()** are identical to the parameters used in **ext_int_install()**. **cs_int_install()** behaves identically to **ext_int_install()**, with the exception that *card_id* must be valid.

Before **cs_card_status()** is called, **cs_init()** must be called to initialize the Card Services/Enabler (CS/Enabler) software layer.

If successful, **cs_int_install()** returns 0. Otherwise, **cs_int_install()** returns `-1` and sets *errno*.

Errors

[EINVAL]	<i>card_id</i> is invalid or the CS/Enabler software layer has not been initialized.
----------	--

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **cs_init()**, p. 141
- **cs_int_query()**, p. 144
- **ext_int_install()** in the OS Open platform-specific user's guides.
- Chapter 11, "OS Open PCMCIA Support," in *OS Open User's Guide*

cs_int_query()

Synopsis

```
#include <sys/csLib.h>
int cs_int_query( int card_id, int event, flih_t *flih );
```

Library

csLib.a

Description

cs_int_query() returns information about the first level interrupt handler (FLIH), if any, for the event named by *event* used by the PC Card identified by *card_id*.

cs_int_query() is used instead of **ext_int_query()** when multiple PC Cards share an OS Open event.

The *event* and *flih* parameters for **cs_int_query()** are identical to the parameters used in **ext_int_query()**. **cs_int_query()** behaves identically to **ext_int_query()**, with the exception that *card_id* must be valid.

Before **cs_card_status()** is called, **cs_init()** must be called to initialize the PCMCIA Card Services/Enabler (CS/Enabler) software layer.

If successful, **cs_int_query()** returns 0. Otherwise, **cs_int_query()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>card_id</i> is invalid or the CS/Enabler software layer has not been initialized.
----------	--

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **cs_init()**, p. 141
- **cs_int_install()**, p. 143
- **ext_int_query()** in the OS Open platform-specific user's guides
- Chapter 11, "OS Open PCMCIA Support," in *OS Open User's Guide*

Synopsis

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

int cs_pata_parm_init( char *pata_name, event_t pata_event,
                      int pata_ISA_int, int install_fat, int fat_dh,
                      char *fat_name, int unit_no, int offset );
```

Library

pataLib.a

Description

cs_pata_parm_init() initializes configurable parameters, in the **cs_card_info** structure *pata_info*, used by the PCMCIA ATA/IDE (PATA) device-specific enabler provided in **pataLib.a**. The parameters are not validated.

pata_info is defined in **<sys/pataLib.h>**. The **cs_card_info** structure is defined in **<sys/csLib.h>**.

pata_name is a pointer to the name that is bound to the PATA device. This pointer is passed to **device_install()** when a PATA device is inserted.

pata_event specifies the OS Open event used by the PATA device driver.

pata_ISA_int is the ISA interrupt routing (number) associated with the event.

If *install_fat* is non-zero, the PATA enabler configures the PATA device into a FAT file system, using the following FAT file system parameters:

- *fat_dh*, the FAT file system device handle obtained from **driver_install()**.
- *fat_name*, a pointer to the name of the FAT file system used to access files on the PATA device.
- *unit_no*, the unit number used by the FAT file system for the PATA device.
- *offset*, the offset of the start of the FAT file system partition on the PATA device.

If *install_fat* is 0, the PATA enabler does not configure the PATA device into the FAT file system and ignores the FAT file system parameters.

cs_pata_parm_init() must be called before the address of the *pata_info* structure is passed to **cs_init()**.

If successful, **cs_pata_parm_init()** returns 0.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	No

cs_pata_parm_init()

References

- **cs_init()**, p. 141
- Chapter 10, “Using the DOS Logical File System,” in *OS Open User's Guide*
- Chapter 11, “OS Open PCMCIA Support,” in *OS Open User's Guide*

Synopsis

```
#include <sys/csLib.h>
unsigned char *cs_set_ccr_offset( int card_id, int offset )
```

Library

csLib.a

Description

cs_set_ccr_offset() passes the offset of a newly inserted card, specified by *card_id*, to the PCMCIA Card Services/Enabler (CS/Enabler) software layer. The offset is from the start of the CIS to the start of the card configuration registers.

cs_set_ccr_offset() allocates a memory window, if necessary to access the configuration registers, and returns the system address to use to access the configuration registers.

If the specified card is not inserted, **cs_set_ccr_offset()** returns -1.

If *ccr_offset* in the **cs_card_info_struct** associated with a PC Card is statically initialized to -1 and the card shares interrupts with other PC Cards, the card's enabler software should call **cs_set_ccr_offset()** whenever the card is inserted.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **cs_init()**, p. 141
- Chapter 11, "OS Open PCMCIA Support," in OS Open User's Guide

cs_stateless_card_status()

Synopsis

```
#include <sys/csLib.h>
int cs_stateless_card_status( int card_id );
```

Library

csLib.a

Description

cs_stateless_card_status() returns information about the insertion state of the PC Card specified by *card_id*.

cs_stateless_card_status() returns 0 if the specified PC Card is plugged into the system and was inserted before the most recent call to **cs_card_status()**.

cs_stateless_card_status() returns 1 if the card is plugged in, but was inserted after the most recent call to **cs_card_status()**. If the specified PC Card is not plugged into the system, or if *card_id* is invalid, **cs_stateless_card_status()** returns -1.

Before **cs_stateless_card_status()** is called, **cs_init()** must be called to initialize the PCMCIA Card Services/Enabler software layer.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **cs_card_status()**, p. 138
- **cs_init()**, p. 141
- Chapter 11, "OS Open PCMCIA Support," in OS Open *User's Guide*

Synopsis

```
#include <time.h>
char *ctime( const time_t *time );
```

Library

cLib.a

Description

ctime() converts the time value pointed to by *time* to local time in a character string and returns a pointer to the resulting string.

A time value, usually obtained using **time()**, is converted into a 26-character string with constant-width fields as illustrated in the following example:

```
Mon Jul 16 02:03:55 1987\n\0
```

ctime() uses a 24-hour-clock. Days and months are abbreviated with their first three letters, beginning with a capital. Newline (\n) and NULL (\0) end the string.

ctime() is functionally equivalent to:

```
asctime(localtime(&anytime))
```

Note: **asctime()** and **ctime()** each store results in a shared statically allocated buffer. A call to either function destroys the result of a previous call.

Errors

None.

Example

See **memheap_query()**, p. 410.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.12.3.2
- **asctime()**, p. 62
- **gmtime()**, p. 299
- **localtime()**, p. 388
- **time()**, p. 886

ctime_r()

Synopsis

```
#include <time.h>
char *ctime_r( const time_t *time, char *buf );
```

Library

cLib.a

Description

ctime_r() is a reentrant version of **ctime()**.

ctime_r() converts the time value pointed to by *time* to local time in a character string and returns a pointer to a buffer that contains the resulting string.

Unlike **ctime()**, **ctime_r()** does not store its results in a statically allocated buffer. Instead, **ctime_r()** stores its results in a programmer-supplied buffer pointed to by *buf*. The buffer must contain space for at least 26 characters.

If unsuccessful, **ctime_r()** returns a NULL.

Errors

None.

Example

See **shm_open()**, p. 734.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §19.2.3.5
- **ctime()**, p. 149

Synopsis

```
#include <dbLib.h>

int dbbrkpt_clr( pthread_t thread, unsigned long address );
```

Library

dbLib.a

Description

dbbrkpt_clr() removes one or more breakpoints.

If *thread* and *address* are 0, all breakpoints are removed. If *thread* is 0 and *address* is not 0, all breakpoints at *address* are cleared. If *thread* is not 0 and *address* is 0, all breakpoints in *thread* are cleared. If both *thread* and *address* are not 0, only one breakpoint is cleared.

If no breakpoint is found, **dbbrkpt_clr()** returns -1. Otherwise, **dbbrkpt_clr()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

dbbrkpt_cont()

Synopsis

```
#include <dbLib.h>
int dbbrkpt_cont( char *cmd, char *buffer );
```

Library

dbLib.a

Description

dbbrkpt_cont() continues after encountering a breakpoint.

A command list associated with a breakpoint is copied into the buffer pointed to by *cmd*.

Calling **dbbrkpt_cont()** generates a message. If *buffer* is NULL, the message is written to the standard output. Otherwise, the message is stored in *buffer*.

dbbrkpt_cont() always returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

Example

See **dbbrkpt_next_inst()**, p. 157.

Synopsis

```
#include <dbLib.h>
void dbbrkpt_disp( char *buf );
```

Library

dbLib.a

Description

dbbrkpt_disp() displays breakpoint status information.

If *buf* is NULL, **dbbrkpt_disp()** writes its output to the standard output. Otherwise, the output is stored in *buf*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

dbbrkpt_find_inst()

Synopsis

```
#include <dbLib.h>
unsigned long dbbrkpt_find_inst( unsigned long address );
```

Library

dbLib.a

Description

dbbrkpt_find_inst() returns the instruction at the address of a breakpoint.

If there is no breakpoint at *address*, **dbbrkpt_find_inst()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <dbLib.h>
int dbbrkpt_init( mqd_t *mqd );
```

Library

dbLib.a

Description

dbbrkpt_init() initializes breakpoint data and installs the Trap and Error first level interrupt handlers (FLIHs).

Note: Programs must call **dbbrkpt_init()** before calling any breakpoint functions (those starting with “dbbrkpt_”).

Note: The variable pointed to by *mqd* is updated with the message queue descriptor that receives breakpoint and error messages.

If successful, **dbbrkpt_init()** returns 0. Otherwise, **dbbrkpt_init()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

dbbrkpt_isbp()

Synopsis

```
#include <dbLib.h>
int dbbrkpt_isbp( pthread_t thread, unsigned long addr );
```

Library

dbLib.a

Description

dbbrkpt_isbp() returns true if a permanent breakpoint is set for the thread specified by *thread* at address *addr*.

Otherwise, **dbbrkpt_isbp()** returns false.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <dbLib.h>

int dbbrkpt_next_inst( pthread_t thread, unsigned long address,
                      unsigned long *target1, unsigned long *target2,
                      Pthread_POWER_reg_t *regp, int next );
```

Library

dbLib.a

Description

dbbrkpt_next_inst() determines the location that the thread specified by *thread* will execute next and copies that location into the object pointed to by *target1*.

If **dbbrkpt_next_inst()** cannot determine just one location, the function copies the alternate location into the object pointed to by *target2*. Otherwise, the object pointed to by *target2* is set to 0.

If register values are needed to determine the next location, **dbbrkpt_next_inst()** reads the buffer pointed to by *regp*. If *regp* is NULL, **dbbrkpt_next_inst()** reads the control block of *thread*. The buffer pointed to by *regp* should be of type **Pthread_POWER_reg_t**, as defined in the file **<dbLib.h>**.

The parameter *next* should be set to one of the following values:

DB_STEP	On subroutine calls, return the location of the called function.
DB_NEXT	On subroutine calls, return the location where the function returns.

If successful, **dbbrkpt_next_inst()** returns 0. Otherwise, **dbbrkpt_next_inst()** returns -1.

Errors

None.

Example

The following example illustrates a function that steps a thread one instruction and prints whether a branch was taken.

```
#include <kadtLib.h>
#include <dbLib.h>
#include <pthread.h>
#include <sys/reg.h>

void step_one(pthread_t thread) {
    unsigned long iar, target1, target2;
    brkpt_cntl_t *sibling;
```

dbbrkpt_next_inst()

```
char string[10];

iar = reg_get(IAR, thread);
dbbrkpt_next_inst(thread, iar, &target1, &target2, NULL, DB_STEP);
dbbrkpt_tset(thread, target1, "no branch", NULL, &sibling);
if (target2 != 0)
    dbbrkpt_tset(thread, target2, "branch", sibling, NULL);
dbbrkpt_cont(string, NULL);
printf("%s taken\n", string);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <dbLib.h>

int dbbrkpt_set( pthread_t thread, unsigned long addr, char *cmdlist );
```

Library

dbLib.a

Description

dbbrkpt_set() sets a breakpoint for the thread specified by *thread* and *addr*.

If *thread* is set to 0, the breakpoint is valid for any thread. The argument *cmdlist* points to a string that **dbbrkpt_cont()** copies when the breakpoint is reached.

If arguments are not valid or a breakpoint was set for *thread* or *addr*, **dbbrkpt_set()** returns -1. Otherwise, **dbbrkpt_set()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **dbbrkpt_cont()**, p. 152

dbbrkpt_tclr()

Synopsis

```
#include <dbLib.h>
int dbbrkpt_tclr( pthread_t thread, unsigned long addr );
```

Library

dbLib.a

Description

dbbrkpt_tclr() clears a temporary breakpoint set by **dbbrkpt_tset()**.

If no temporary breakpoint is found, **dbbrkpt_tclr()** returns -1. Otherwise, **dbbrkpt_tclr()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <dbLib.h>

int dbbrkpt_tset( pthread_t thread, unsigned long addr, char *cmdlist,
                  brkpt_cntl_t *sibling, brkpt_cntl_t **id_p );
```

Library

dbLib.a

Description

dbbrkpt_tset() sets a temporary breakpoint at *addr* that is cleared after it is reached by the thread specified by *thread*.

This type of breakpoint is used to implement the **stepi** and **nexti** debugger commands.

If *id_p* is not NULL, **dbbrkpt_tset()** copies a value to the location pointed to by *id_p*. That location can be specified by *sibling* on a subsequent **dbbrkpt_tset()** call. Using a sibling breakpoint allows one call to **dbbrkpt_tclr()** to clear both breakpoints.

The argument *cmdlist* points to a character string that is copied when **dbbrkpt_cont()** is called.

If arguments are not valid or a temporary breakpoint is already set for *thread* or *addr*, **dbbrkpt_tset** returns -1. Otherwise, **dbbrkpt_tset()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Example

See **dbbrkpt_next_inst()**, p. 157.

dbbrkpt_write_inst()

Synopsis

```
#include <dbLib.h>
void dbbrkpt_write_inst( unsigned long address,
                        unsigned long instruction );
```

Library

dbLib.a

Description

dbbrkpt_write_inst() writes *instruction* to *address* and synchronizes the instruction and data caches.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

Synopsis

```
#include <dbLib.h>

int dbmem_disp( unsigned long startaddr, unsigned int count, char *buf );
```

Library

dbLib.a

Description

dbmem_disp() displays the contents of memory starting at *startaddr* until *count* bytes are displayed.

If *buf* is NULL, the output is written to the standard output. Otherwise the information is stored in *buf*.

The resulting display shows the memory in hexadecimal and ASCII and suppresses more than two identical lines of data.

If *count* is less than 1, **dbmem_disp()** returns -1. If successful, **dbmem_disp()** returns 0.

Note: **dbmem_disp()** does not check address validity.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

dbmem_listi()

Synopsis

```
#include <dbLib.h>
int dbmem_listi( unsigned int startaddr, unsigned int count, char *buf );
```

Library

dbLib.a

Description

dbmem_listi() disassembles memory starting at *startaddr* for *count* instructions.

If *buf* is NULL, the output is written to the standard output. Otherwise the output is stored in *buf*.

If *startaddr* is not on an even fullword boundary or *count* is 0, **dbmem_listi()** returns -1. Otherwise, **dbmem_listi()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

Synopsis

```
#include <dbLib.h>
int dbstepi( int next, pthread_t thread, int count );
```

Library

dbLib.a

Description

dbstepi() steps *thread* by *count* instructions.
The parameter *next* can take the following values:
DB_STEP Step through subroutine calls
DB_NEXT Step over subroutine calls
If *thread* or *next* is not valid, **dbstepi()** returns -1. Otherwise, **dbstepi()** returns 0.
Note: OpenShell must have been started.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

dbsymname_find()

Synopsis

```
#include <dbLib.h>
```

```
unsigned long dbsymname_find( unsigned long addr, char *symn );
```

Library

dbLib.a

Description

dbsymname_find() copies the name of the first symbol preceding *addr* into the buffer pointed to by *symn*, and returns the offset from the symbol to *addr*.

Note: The buffer pointed to by *symn* should contain at least 40 characters.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <dbLib.h>

int dbwhere( pthread_t thread, char *buf );
```

Library

dbLib.a

Description

dbwhere() displays stack trace-back information that shows the sequence of calls made by *thread*.

If *buf* is NULL, the output is written to the standard output. Otherwise, the information is stored in the buffer pointed to by *buf*.

If *thread* is not a valid thread, **dbwhere()** returns -1. Otherwise, **dbwhere()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

dcache_flush()

Synopsis

```
#include <ioLib.h>

void dcache_flush( void *address, unsigned int count );
```

Library

ioLib.a

Description

dcache_flush() flushes cache blocks, beginning at the effective address and continuing for *count* bytes. Consult the reference manual for the specific PowerPC processor to determine the cache block size.

A cache line flush forces the current contents of the cache block to main storage (if the block is valid and marked as modified) and then invalidates the block.

Note: Since cache flushes occur on cache block boundaries, the operation can occur outside of the bounds specified by the function call. For example, on a 603 processor, if *address* is X'216' and *count* is X'12', two cache blocks, spanning addresses from X'200' to X'23F', would be flushed.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **dcache_invalidate()**, p. 169

Synopsis

```
#include <ioLib.h >
```

```
void dcache_invalidate( void *address, unsigned int count );
```

Library

ioLib.a

Description

dcache_invalidate() invalidates cache blocks beginning at the effective address given by *address* and continuing for *count* bytes. Consult the reference manual for the specific PowerPC processor to determine the cache block size.

Note: Since cache invalidation occurs on cache block boundaries, invalidation can occur outside of the bounds implied by this command. For example, on a 603 processor, if *address* is X '104' and *count* is 16, the cache block spanning addresses from X '100' to X '120' would be invalidated.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **dcache_flush()**, p. 168

decompress()

Synopsis

```
#include <comprLib.h >
int decompress (char *arguments)
```

Library

comprLib.a

Description

decompress() decompresses data compressed using a UNIX compress command. Compressed data must contain three-bytes of header information. The **arguments* parameter points to a string of the following form:

```
source_file  target_file
└──────────┴────────────────────────────────────────┘
-mem source_address  destination_address  source_size
```

source_file	Name of the file containing compressed data.
target_file	Name of the file that will contain uncompressed data.
-mem	Specifies that compressed data is located in memory at <i>source_address</i> .
destination_address	Specifies the target memory location for decompressed data.
source_size	Specifies the size of compressed data in memory.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

Synopsis

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

int dev_io_init( int fd_max, int dvr_max );
```

Library

devLib.a

Description

dev_io_init() prepares the device subsystem for use and initializes the NULL and shared memory device drivers.

dev_io_init() must successfully complete before any input/output (I/O) functions are available. Normally, **dev_io_init()** is invoked once during system initialization to prepare the device subsystem for use. Application designers must ensure that **dev_io_init()** completes successfully.

The argument *fd_max* specifies the maximum number of files that can be open at one time. If the value of *fd_max* is less than **POSIX_OPEN_MAX**, the value of **POSIX_OPEN_MAX** is used.

The argument *dvr_max* specifies the number of device drivers than can be identified to the device subsystem. Specify one for each distinct type of driver that is installed by the application.

If successful, **dev_io_init()** returns 0 and sets **sysconf(_SC_OPEN_MAX)** to the value of *fd_max*. Otherwise, **dev_io_init()** returns -1 and sets *errno*.

Errors

[EINVAL]	The value of <i>fd_max</i> or <i>dvr_max</i> is negative.
[ENOMEM]	Insufficient memory.

Example

See **read()**, p. 633.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **device_install()**, p. 172
- **device_uninstall()**, p. 174
- **driver_install()**, p. 177

device_install()

Synopsis

```
#include <sys/devLib.h>
#include <sys/devDriver.h>

int device_install( const char *name, int dev_type, int dev_handle, ... );
```

Library

devLib.a

Description

device_install() associates a device driver with a device name in the root directory managed by the device subsystem.

device_install() calls a device-specific configuration function that allows the driver to initialize state and control information for a named instance of itself.

name points to the name that identifies the device on subsequent **open()** calls. *name* appears in the file system name space and must be unique for device special files. *name* should begin with **/dev/**, but the device subsystem does not enforce this convention. *dev_handle*, returned by a preceding **driver_install()** call, identifies the driver associated with *name*.

Note: *name* should come from the portable file name set and should not exceed system restrictions for name and path length. **device_install()** does not enforce the restrictions. However, if *name* does not meet these naming requirements, applications cannot open the device as a file.

Installed devices can be one of the following types:

CHRTYPE	Character special file (serial ports, communication lines)
BLKTYPE	Block special file (random access media)
DTYPE_TTY	Terminal device
LFSTYPE	Logical file system (drivers providing file system services)

A physical device can present multiple interfaces. Consult device driver documentation to determine the value of *type*.

The device subsystem passes the remaining variable arguments to the device-specific configuration function without examination.

Note: Attempting to simultaneously install multiple devices of the same name from multiple threads causes unpredictable results. The device subsystem does not serialize calls to the device-specific configuration function, which must be async safe.

If successful, **device_install()** returns 0. Otherwise, **device_install()** returns -1 and sets *errno*.

Errors

[EEXIST]	The device with <i>name</i> already exists in the device configuration.
[ENOMEM]	Insufficient memory is available to allocate internal control structures.
[EIO]	The driver-specific configuration function returned with a non-zero return code.

Example

See **read()**, p. 633.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **dev_io_init()**, p. 171
- **device_uninstall()**, p. 174
- **driver_install()**, p. 177

device_uninstall()

Synopsis

```
#include <sys/devLib.h>
int device_uninstall( const char *name );
```

Library

devLib.a

Description

device_uninstall() removes a device that was installed with **device_install()**. The parameter name is the same character string provided with the **device_install()** call.

device_uninstall() can only uninstall a device driver when that device driver supports its removal. When the configure function for the device driver being removed is called, the first variable argument is set to -1 to distinguish the call from a device being added.

If successful, **device_uninstall()** returns 0. Otherwise, **device_uninstall()** returns -1 and sets *errno*.

Errors

ENOENT	The device specified by name could not be found.
EIO	The driver specific configuration function returned with a non-zero return code.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **dev_io_init()**, p. 171
- **device_install()**, p. 172
- **driver_install()**, p. 177

Synopsis

```
#include <time.h>

double difftime ( time_t time1, time_t time0 );
```

Library

cLib.a

Description

difftime() computes the difference between the two calendar times given by *time1* and *time0*.

difftime() returns the difference in seconds.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.12.2.2

div()

Synopsis

```
#include <stdlib.h>

```

Library

cLib.a

Description

div() calculates the quotient and remainder of the division of *numerator* by *denominator*.

div() returns a structure of type **div_t** that contains an integer quotient, *quot*, and integer remainder, *rem*.

If the returned value cannot be represented, its value is undefined.

Errors

None.

Example

See **atoi()**, p. 72.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.6.2
- **ldiv()**, p. 355

Synopsis

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

int driver_install( int *dev_handle, int(*init_fcn)(), ... );
```

Library

devLib.a

Description

driver_install() identifies a device driver to the device subsystem.

The argument *init_fcn* is a function provided by the designer of the device driver that initializes the internal state, control areas, and so on, of the device driver. Variable arguments, if any, are passed to the initialization function without examination by the device subsystem. Refer to device driver documentation for the format of any required variable arguments.

driver_install() generally calls *init_fcn* once for each device driver to be installed. Because a successful call establishes a device driver entry in the device subsystem, the value of *dvr_max* supplied to **dev_io_init()** must at least match the number of **driver_install()** calls.

If successful, **driver_install()** returns 0 and sets *dev_handle* for use in a subsequent **device_install()** function call. Otherwise, **driver_install()** returns -1 and sets *errno*.

Errors

[ENOSPC]	There are no free device driver slots. Increase <i>dvr_max</i> in the preceding dev_io_init() call to match the number of desired device drivers.
[EIO]	<i>init_fcn</i> returned a nonzero return code.

Example

See **read()**, p. 633.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **dev_io_init()**, p. 171
- **device_install()**, p. 172
- **device_uninstall()**, p. 174

dtom()

Synopsis

```
#include <sys/mbuf.h>
#include <sys/types.h>
struct mbuf *dtom( x );
```

Library

None. This is a macro defined in `<sys/mbuf.h>`.

Description

dtom() converts the address stored in *x*, which points into the data portion of a memory buffer (mbuf), to the address of the first byte of the same mbuf.

dtom() uses the facts that mbufs are of a fixed size, are allocated contiguously, and are aligned on **sizeof(mbuf)** boundaries.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **mtod()**, p. 458

Synopsis

```
#include <netinet/if_ether.h>
#include <sys/mbuf.h>

void enet_arpinput( struct arpcom *ac, struct mbuf *m );
```

Library

tcpiplib.a

Description

enet_arpinput() provides an Ethernet address resolution protocol (ARP) input routine.

The argument *ac* points to a structure, **arpcom**, shared by the Ethernet device driver and ARP routines. The argument *m* points to a buffer that holds a raw Ethernet ARP packet.

The Ethernet device driver should call **enet_arpinput()** when it receives an ARP packet.

Note: **enet_arpinput()** supports standard Ethernet only; 802.3 Ethernet is not supported.

Errors

None.

Example

The following example passes raw internet protocol (IP) ARP packets to **enet_arpinput()**:

```
struct mbuf *m;
unsigned long n;
struct arpcom enet_arp_if;
:
:
case ETHERTYPE_ARP:
    n=sizeof(struct ether_header);
    m->m_data+=n;
    m->m_len-=n;
    m->m_pkthdr.len-=n;
    (void)enet_arpinput(&enet_arp_if, m);
    break;
:
:
```

enet_arpinput()

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- `enet_arpresolve()`, p. 181

Synopsis

```
#include <netinet/if_ether.h>
#include <sys/mbuf.h>

int enet_arresolve( struct arpcom *ac, struct mbuf *m,
                   struct in_addr *destip, unsigned char *desten );
```

Library

tcipLib.a

Description

enet_arresolve() resolves an Internet Protocol (IP) address to an Ethernet address.

If the resolution succeeds, the argument *desten* is filled in with an Ethernet destination address. The argument *destip* is a filled structure containing the destination IP address. The argument *m* contains the packet to be transmitted.

If the destination address is the same as the local address, the loopback interface is used for transmission when available. If **enet_arresolve()** cannot immediately resolve the address, the function transmits an address resolution protocol (ARP) packet. The packet *m* is queued for transmission when the address is resolved.

This routine is called by the Ethernet device driver output routine.

Note: **enet_arresolve()** supports standard Ethernet only; 802.3 Ethernet is not supported.

If the IP address was resolved and *m* can be transmitted immediately, **enet_arresolve()** returns 1. If the address cannot be resolved immediately, **enet_arresolve()** returns 0.

Errors

None.

Example

The following example passes an (IP) packet to **enet_arresolve()** for address resolution:

```
struct mbuf    *m;
struct arpcom  *acp;
struct sockaddr *dst;
unsigned char[6] enet_dest;
struct in_addr idst;
struct ether_header eh;
:
.
```

enet_arpresolve()

```
switch (dst->sa_family)
case AF_INET:
    if (!enet_arpresolve(acp, m, &idst, enet_dest)) {
        return(0);
    }
    eh.ether_type=ETHERTYPE_IP;
    break;
    :
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- `enet_arpinput()`, p. 179

Synopsis

```
#include <netinet/if_ether.h>
int enet_attach( char *enet_file );
```

Library

netLib.a

Description

enet_attach() connects an Ethernet interface to TCP/IP protocol stacks.

The argument *enet_file* contains the file name of the installed Ethernet device driver.

Note: **enet_attach()** supports only the standard Ethernet device driver.

If successful, **enet_attach()** returns 0. Otherwise, **enet_attach()** returns -1.

Errors

None.

Example

The following example sets up an Ethernet interface for TCP/IP:

```
#include <sys/devLib.h>
#include <sys/devDriver.h>

int devhandle;
int enet_init(driver_t *dsw, va_list varg);

if (driver_install(&devhandle, enet_init, 5, 256, 128)) {
    (void)printf("driver_install() failed\n");
    return(-1);
}
if (device_install("/dev/en0", CHRTYPE, devhandle)) {
    (void)printf("device_install() failed\n");
    return(-1);
}
if (tcpip_init("myhost", 1, 200)) {
    (void)printf("tcpip_init() failed\n");
    return(-1);
}
if (net_init()) {
    (void)printf("net_init() failed\n");
    return(-1);
}
if (enet_attach("/dev/en0")) {
    (void)printf("enet_attach() failed\n");
}
```

enet_attach()

```
        return(-1);
    }
    if (ifconfig("ent0 myhost netmask 255.255.240.0")) {
        (void)printf("ifconfig() failed\n");
        return(-1);
    }
    (void)ping("brightblade 4096 10");
```

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- `if_attach()`, p. 303

Synopsis

```
#include(sys/devLib.h)
int errlog( char *buffer, unsigned int cnt );
```

Library

devLib.a

Description

errlog() writes an error string to the error log device driver.

buffer points to a character string containing a user-defined error message. *cnt* specifies the size in bytes of the error message pointed to by *buffer*.

This function attempts to write the message to the user-supplied **/dev/error** device, if it exists. In any case, the first 96 bytes of the error message are stored internally in the device I/O support library dsa area. This area can be located by examining the associated pointer for the **devLib.a** package obtained from **library_list()**. In this way, the last error message can be obtained from memory even if no error logging device has been defined.

Errors

None

Example

```
void aerologist()
{
    char er_test[] = "error log test line";
    errlog(er_test, strlen(er_test) + 1); /* insure trailing null */
    /* The pointer value in the printf() statement following this block comment
    is the value of the devLib dsa. This may be determined in two ways.
    * Examine the load map for the label "dsa"
    * Use the library_list() function to find the associated value for devLib.a
    An example of library_list() output is below:
```

```
Installed packages Anchor/ Package name
0x0  shell.o  1.2 12/5/94
0xba7f0  fatLib.a  1.26 3/29/95
0x3ffcdc  ramdLib.a  1.2 1/27/95
0x3b9fb0  fsLib.a  1.1 7/28/94
0x0  ttyLib.a  1.2 12/15/94
0xa53c0  asyncLib.a  1.6 3/9/95
```

errlog()

0xa3430 devLib.a 1.6 4/12/95

```
-----  
*/  
    printf("Error line retrieved from memory: %s\n", 0xa3430);  
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

Synopsis

```
#include <stdlib.h>
char *exit( int status );
```

Library

None. This is a macro defined in **<stdlib.h>**.

Description

exit() calls **pthread_exit()**, which terminates the calling thread.

exit() is provided for ANSI C compatibility.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.10.4.3
- **pthread_exit()**, p. 557

exp()

Synopsis

```
#include <math.h>
double exp( double x );
```

Library

mathLib.a

Description

exp() computes the exponential function of *x*.

If the magnitude of *x* is too large, a range error occurs and **exp()** sets *errno*.

Errors

[ERANGE]	The magnitude of <i>x</i> is too large.
----------	---

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.4.1

Synopsis

```
#include <math.h>
double fabs( double x );
```

Library

mathLib.a

Description

fabs() returns the absolute value of the floating-point number *x*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.6.2

fclose()

Synopsis

```
#include <stdio.h>
int fclose( FILE *stream );
```

Library

fsLib.a

Description

fclose() closes a stream pointed to by *stream*, flushing all buffers associated with *stream* before closing it.

When **fclose()** closes *stream*, the function releases any buffers reserved by OS Open.

If successful, **fclose()** returns 0. If *stream* was already closed, or if an error occurs, **fclose()** returns EOF.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **fopen()**, p. 210.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.5.1
- **fs_init()**, p. 231

Synopsis

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

int fcntl( int filedes, int cmd, ... )
```

Library

devLib.a

Description

fcntl() provides control commands for open files.

The argument *filedes* is a file descriptor.

The available values for *cmd*, defined in the header file **<fcntl.h>**, include:

F_DUPFD	Returns a new file descriptor that duplicates <i>filedes</i> and refers to the same open file description. The value of the new descriptor must be greater than or equal to a third argument, <i>arg</i> , taken as type int .
F_GETFD	Gets the file descriptor flags associated with <i>filedes</i> . The flags are associated with only one file descriptor; they do not affect other file descriptors that refer to the same file.
F_SETFD	Sets the file descriptor flags associated with <i>filedes</i> to the third argument, <i>arg</i> , taken as type int . Gets file status flag values and file access mode values for the open file description associated with <i>filedes</i> . The file access modes can be extracted from the return value using the mask O_ACCMODE defined in the file <fcntl.h> . File status flags and file access modes are associated with the open file description and do not affect other file descriptors that refer to the same file using different open file descriptions.
F_SETFL	Set the file status flag values for the open file description associated with <i>filedes</i> using the corresponding bits in the third argument, <i>arg</i> , taken as type int . Bits corresponding to the file access mode and the <i>oflag</i> values set in <i>arg</i> are ignored. If the calling program changes other bits in <i>arg</i> , results are unspecified

cmd constants for **fcntl()** follow:

fcntl()

F_DUPFD	Duplicate file descriptor
F_GETFD	Get file descriptor tags
F_SETFD	Set file descriptor flags
F_GETFL	Get file status flags
F_SETFL	Set file status flags

File status flag constants for **fcntl()** and **open()** follow:

O_APPEND	Set append mode
O_NONBLOCK	No delay

File access mode constants for **fcntl()** and **open()** follow:

O_RDONLY	Open read-only
O_RDWR	Open read/write
O_WRONLY	Open write-only

If successful, **fcntl()** returns one of the following values, depending on *cmd*. Otherwise, **fcntl()** returns `-1` and sets *errno*.

F_DUPFD	New file descriptor
F_GETFD	Non-negative value of file descriptor flags
F_SETFD	Value other than <code>-1</code>
F_SETFL	Non-negative value of file status flags and access modes

Errors

[EBADF]	<i>filedes</i> is not a valid file descriptor.
[EINVAL]	<i>cmd</i> is F_DUPFD , and <i>arg</i> is negative or greater than or equal to {OPEN_MAX} .
[EMFILE]	<i>cmd</i> is F_DUPFD and {OPEN_MAX} file descriptors are currently in use by this process, or no file descriptors greater than or equal to <i>arg</i> are available.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §6.5.2
- **close()**, p. 127
- **open()**, p. 478

Synopsis

```
#include <stdio.h>
FILE *fdopen( int fd, const char *type );
```

Library

fsLib.a

Description

fdopen() opens a stream and associates it with the file descriptor *fd*.

The argument *type* is a character string specifying the access requested for the file. *type* contains one positional parameter; its possible values follow:

Parameter	Description
r	Open a file for reading.
w	Open a file for writing.
a	Open a file for writing at end-of-file (EOF).
r+	Open a file for update (reading and writing).
w+	Open a file for update (reading and writing).
a+	Open a file for update (reading and writing) at EOF.

When a file is opened with **a** or **a+**, all write operations are at EOF. Although a call to **fseek()** or **rewind()** can reposition the file position indicator, a write operation moves the file position indicator to EOF before writing the file. Thus, you cannot write over existing data.

When switching between reading and writing in update mode, you must include an intervening position function call, such as **fseek()**, **fsetpos()**, **rewind()**, or **fflush()**. Output can follow input without an intervening call if EOF was detected.

fdopen() returns a pointer to a file structure that can access the open file. A NULL pointer indicates an error.

Note: The calling program must ensure that the mode of the stream is allowed by the mode of the open file.

Note: The file system must have been initialized by **fs_init()**.

Errors

[ENOTSUP]	fs_init() has not initialized the file system.
[EINVAL]	<i>type</i> is not valid.
[ENOMEM]	Not enough memory is available to open the file
[EMFILE]	The maximum number of opened files was reached.
[EBADF]	<i>fd</i> is not valid.

fdopen()

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §8.2.2
- **fopen()**, p. 210
- **fs_init()**, p. 231

Synopsis

```
#include <stdio.h>

int feof( FILE *stream );
```

Library

fsLib.a

Description

feof() tells whether the end-of-file (EOF) flag is set for *stream*.

Several functions set the EOF flag to indicate EOF.

To reset the EOF flag, close *stream* or call **rewind()**, **fsetpos()**, **fseek()**, or **clearerr()** for *stream*.

If the EOF flag is set, **feof()** returns a nonzero value. Otherwise, **feof()** returns 0.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.9.10.2
- **clearerr()**, p. 99
- **ferror()**, p. 196
- **fs_init()**, p. 231
- **perror()**, p. 488

ferror()

Synopsis

```
#include <stdio.h>
int ferror( FILE *stream );
```

Library

fsLib.a

Description

ferror() tests for an error in reading from or writing to the stream pointed to by *stream*.

If an error occurs, the error indicator for *stream* remains set until *stream* is closed, or **rewind()** or **clearerr()** is called.

If an error occurs on *stream*, **ferror()** returns a nonzero value. Otherwise, **ferror()** returns 0.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **fopen()**, p. 210.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.9.10.3
- **clearerr()**, p. 99
- **feof()**, p. 195
- **fopen()**, p. 210
- **fs_init()**, p. 231
- **perror()**, p. 488

Synopsis

```
#include <stdio.h>
int fflush( FILE *stream );
```

Library

fsLib.a

Description

fflush() empties the buffer associated with the output stream pointed to by *stream*.

If *stream* is open for input only, **fflush()** returns EOF; *stream* remains open after the call.

If *stream* is a NULL pointer, the system flushes all streams opened for write.

The **fflush()** function returns the value 0 if it successfully flushes the buffer. **fflush()** returns EOF if a write error occurs.

Note: OS Open automatically flushes buffers when they become full, or when a stream closes. The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **fread()**, p. 225.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.5.2
- **fclose()**, p. 190
- **fs_init()**, p. 231
- **setbuf()**, p. 720

fgetc()

Synopsis

```
#include <stdio.h>
int fgetc( FILE *stream );
```

Library

fsLib.a

Description

fgetc() returns the next character from the input stream pointed to by *stream* and advances the file position indicator.

An EOF return value indicates an error or end-of-file condition (EOF). Use **feof()** or **ferror()** to determine whether EOF indicates an error or EOF.

fgetc() is identical to **getc()**, but **getc()** is a macro; **fgetc()** is a function.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.7.1
- **feof()**, p. 195
- **ferror()**, p. 196
- **fgetc()**, p. 198
- **fputc()**, p. 222
- **fs_init()**, p. 231

Synopsis

```
#include <stdio.h>

int fgetpos( FILE *stream, fpos_t *pos );
```

Library

fsLib.a

Description

fgetpos() stores the value of the file position indicator of the stream pointed to by *stream* in the object pointed to by *pos*.

A subsequent call to **fsetpos()** can use the value pointed to by *pos* to reposition *stream*'s file position indicator.

If successful, **fgetpos()** returns 0. Otherwise, **fgetpos()** returns a nonzero value and sets *errno*.

Note: The file system must have been initialized by **fs_init()**.

Errors

[EINVAL] *pos* is not valid.

Example

The following example prints a message at end-of-file and restores the previous file position.

```
#include <stdio.h>
#define COMPLETE 0

int ex_fgetpos(char *filename, char *msg)
{
    fpos_t position;
    FILE *fp;

    fp = fopen(filename, "r+");
    printf("file has been opened\n");

    /* Save current file position */
    if(fgetpos(fp, &position) != 0) {
        printf("Unable to save current file position\n");
        return(EEOF);
    }
    /* Change position to end of file */
    printf("Current position has been saved\n");
    if(fseek(fp, 0, SEEK_END) != 0) {
        printf("Unable to change position to EOF\n");
        return(EEOF);
    }
    printf("EOF found\n");
```

fgetpos()

```
/* Write message to file */
if(fputs(msg, fp) == EOF) {
    printf("Unable to write to file\n");
    return(EOF);
}
printf("msg written OK\n");
/* Restore previous file position */
if(fsetpos(fp, &position) != 0) {
    printf("Unable to restore file position\n");
    return(EOF);
}
printf("file position restored\n");
fclose(fp);
return(COMPLETE);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.9.9.1
- **fs_init()**, p. 231
- **fseek()**, p. 235
- **fsetpos()**, p. 237
- **ftell()**, p. 239

Synopsis

```
#include <stdio.h>

char *fgets( char *string,
             int n,
             FILE *stream );
```

Library

fsLib.a

Description

fgets() reads a string from the input stream pointed to by *stream* and stores it in the buffer pointed to by *string*.

fgets() reads from the current *stream* position until a new-line (`\n`) or the end of the stream, or until the number of characters read equals $n - 1$. **fgets()** stores the result in *string*, which it terminates with a NULL (`\0`). If **fgets()** reads a new-line, it is stored in *string*. If n equals 1, *string* is empty.

fgets() returns a pointer to *string*. A NULL return value indicates an error or end-of-file (EOF). Use **feof()** or **ferror()** to determine whether the NULL value indicates an error or EOF.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **fopen()**, p. 210.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.7.2
- **feof()**, p. 195
- **ferror()**, p. 196
- **fputs()**, p. 224
- **fs_init()**, p. 231
- **gets()**, p. 289
- **puts()**, p. 615

fileno()

Synopsis

```
#include <stdio.h>
int fileno( FILE *stream )
```

Library

None. This is a macro defined in **<stdio.h>**.

Description

fileno() returns the file descriptor associated with *stream*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, §8.2.1
- **open()**, p. 478

Synopsis

```
#include <kadtLib.h>
int flih_list( char **bufp );
```

Library

kadtLib.a

Description

flih_list() provides formatted information about all first level interrupt handlers (FLIHs).

The information is part of the **kda_dump()** display.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

flih_list() returns 0.

Errors

None.

Example

The following example shows a sample listing of first level interrupt handlers.

```
OS OPEN>flih_list()
```

Installed Interrupt Handlers	Stack	Arg
EXT0 0x26cefc my_sneak_flih	0x27ad30	0x27abf8
DECREMENTER 0x26cf08 __timer_flih	0x4bfe44	0x1
INVOP 0x26d1a8 dberr_flih	0x45fee0	0x2
TRAP 0x26d19c dbtrap_flih	0x460ae8	0x3
PRINST 0x26d1a8 dberr_flih	0x45fee0	0x4
FPE 0x26d1a8 dberr_flih	0x45fee0	0x5
ALIGNMENT 0x26d1a8 dberr_flih	0x45fee0	0x6
SYSRESET 0x26d1a8 dberr_flih	0x45fee0	0x7
MACHCHK 0x26d1a8 dberr_flih	0x45fee0	0x8
INSTST 0x26d1a8 dberr_flih	0x45fee0	0x9
DATAST 0x26d1a8 dberr_flih	0x45fee0	0xa
NOFPU 0x26dc50 fpunavail_handler	0x172bfc	0x0
TRACE 0x26d1a8 dberr_flih	0x45fee0	0xc
PERFORMANCE 0x26d1a8 dberr_flih	0x45fee0	0xd
IAB 0x26d1a8 dberr_flih	0x45fee0	0xe
SMI 0x26d1a8 dberr_flih	0x45fee0	0xf
DST_MISS 0x26d1a8 dberr_flih	0x45fee0	0x10
DTLB_MISS 0x26d1a8 dberr_flih	0x45fee0	0x11
ITLB_MISS 0x26d1a8 dberr_flih	0x45fee0	0x12

flih_list()

IO_IFC	0x26d1a8 dberr_flih	0x45fee0	0x13
RNMODEXCP	0x26d1a8 dberr_flih	0x45fee0	0x14
----- Total Number of Installed Interrupt Handlers: 21 -----			

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **kda_dump()**, p. 349

Synopsis

```
#include <stdio.h>

void flockfile( FILE *stream );
```

Library

fsLib.a

Description

flockfile() is called by a thread to lock (get exclusive access to) the stream pointed to by *stream*.

If another thread has locked *stream*, the calling thread is blocked until it can get exclusive access.

Note that a single thread can have nested matching calls to **flockfile()** and **funlockfile()** for one stream. If this occurs, only the first call to **flockfile()** and the matching call to **funlockfile()** have any effect.

flockfile() returns nothing.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

The following example function locks two files, reads hexadecimal numbers a character at a time, translates the number into decimal, writes it a character at a time to the second file, and unlocks the files.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stddef.h>
#include <ctype.h>

void ex_flockfile(char *read_file, char *store_file)
{
    unsigned long num;
    char temp_buf[20];
    char *wp;
    FILE *rf, *sf;
    int rc = 0;

    rf = fopen(read_file, "r");
    sf = fopen(store_file, "w");

    /* Lock files for exclusive use */
```

flockfile()

```
flockfile(rf);
flockfile(sf);

printf("Files have been locked for exclusive use\n");
/* Translate numbers until end of file is reached */
while(!feof(rf)) {
    printf("end of file has not been reached\n");
    memset(temp_buf, '\0', 20);
    wp = temp_buf;
    /* read in a number one character at a time */
    *wp = getc_unlocked(rf);
    while(isxdigit(*wp))
    {
        printf("character is %c\n", *wp);
        printf("putting character in the buffer\n");
        wp++;
        *wp = getc_unlocked(rf);
    }
    /* If end of a hexadecimal number, translate number */
    /* and store in other file */
    if(wp != temp_buf)
    {
        printf("end of hexadecimal number found\n");
        num = strtoul(temp_buf, NULL, 16);
        sprintf(temp_buf, "%d", num);
        wp = temp_buf;
        while(*wp != '\0')
        {
            printf("adding number to output file\n");
            rc = putc_unlocked(*wp, sf);
            printf("return code is %d\n", rc);
            wp++;
        }
    }
}

/* Unlock files */
funlockfile(rf);
funlockfile(sf);
fclose(rf);
fclose(sf);
printf("files have been unlocked and closed\n");
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8 §19.2.2.1
- **fs_init()**, p. 231
- **funlockfile()**, p. 252

floor()

Synopsis

```
#include <math.h>
double floor( double x );
```

Library

mathLib.a

Description

floor() returns the largest integral value not greater than *x*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.6.3

Synopsis

```
#include <math.h>
double fmod( double x, double y );
```

Library

mathLib.a

Description

fmod() computes the floating point remainder of x/y .

fmod() returns 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 y is 0, a domain error occurs, and **fmod()** sets *errno* and returns 0.

Errors

[EDOM] y is 0.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.6.4

fopen()

Synopsis

```
#include <stdio.h>

FILE *fopen( const char *filename,
             const char *type );
```

Library

fsLib.a

Description

fopen() opens the file named by the string pointed to by *filename* and associate a stream with the file.

type, a character string that specifies the access requested for the file, contains a positional parameter. The possible values for the positional parameter follow:

Parameter	Description
r	Open an existing text file for reading.
w	Create a text file for writing. If the file exists, its contents are destroyed.
a	Open a text file in append mode. If the file does not exist, fopen() creates it.
r+	Open an existing text file for reading and writing.
w+	Create a text file for reading and writing. If the file exists, its contents are destroyed.
a+	Open a text file in append mode. If the file does not exist, fopen() creates it.
rb	Open an existing binary file for reading.
wb	Create an empty binary file for writing. If the file exists, its contents are destroyed.
ab	Append a binary file for writing. If the file does not exist, fopen() creates it.
r+b or rb+	Open an existing binary file for reading and writing.
w+b or wb+	Create an empty binary file for reading and writing. If the file exists, its contents are destroyed.
a+b or ab+	Open a binary file in append mode. If the file does not exist, fopen() creates it.

fopen() will set the default buffering depending on whether the file is opened as a text or binary file. If the file is opened as binary (a "b" in the *type* string), the

buffering will default to full buffering. If the file is opened as a text file, the buffering will default to line buffering. The default can be changed by calling **setvbuf()** before any reads or writes are done to the file.

Opening a file with one of the **a**, **a+**, **ab**, **a+b**, or **ab+** types causes all write operations to occur at end-of-file (EOF). Although **fseek()** and **rewind()** reposition the file position indicator, a write operation moves the file position indicator to EOF before writing the file. Thus, you cannot overwrite existing data.

Specifying an update mode (using + in the second or third position) allows a program to read from and write to a file. However, a positioning function, such as **fflush()**, **fseek()**, **fsetpos()**, or **rewind()**, must be called between reads and writes. Output may immediately follow input if EOF was detected.

fopen() returns a pointer to a file structure that can be used to access the open file. A NULL pointer return value indicates an error.

Note: The file system must have been initialized by **fs_init()**. For optimal performance with regular files, **setvbuf()** should be used to set the file's buffering to full buffering. **setvbuf()** must be called before any reads or writes to the file have occurred.

Errors

[ENOTSUP]	fs_init() has not initialized the file system.
[EINVAL]	<i>type</i> is not valid.
[ENAMETOOLONG]	<i>filename</i> is too long for a file name.
[EACCES]	<i>filename</i> is already open in a different mode.
[ENOMEM]	Insufficient memory is available to open the file.
[EMFILE]	Maximum number of opened files has been reached.

Example

The following example counts the number of code lines, comment lines, partial code lines, blank lines, and total lines in a file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>

#define ERROR 1
#define COMPLETE 0

int lcc(char *fname)
{
    FILE *fp;
    char *buffer, *wp, *cp;
    unsigned int codelines = 0, comlines = 0, partlines = 0, total = 0;
```

fopen()

```
unsigned int blanklines = 0;
/* allocate working buffer */
buffer = (char *)malloc(300);
memset(buffer, '\0', 300);
/* open file for reading */
fp = fopen(fname, "r");
if(fp == NULL)
{
    printf("Error in opening file %s\n", fname);
    return(ERROR);
}
/* Set buffering to full buffering */
if (setvbuf(fp, NULL, _IOFBF, 0))
{
    printf("Error setting buffering\n");
    fclose(fp);
    free(buffer);
    return(ERROR);
}
/* Read from the file one line at a time */
while(fgets(buffer, 300, fp) != NULL)
{
    /* An error occurred reading from the file */
    if(ferror(fp))
    {
        printf("Error in reading file %s\n", fname);
        fclose(fp);
        free(buffer);
        return(ERROR);
    } /* if(!ferror()) */
    wp = buffer;
    /* increment total line counter */
    total++;
    /* ignore any leading whitespaces */
    while((isspace(*wp)) && (*wp != '\n'))
        wp++;
    /* blank line */
    if(*wp == '\n')
    {
        blanklines++;
        continue;
    }
    /* comment line */
    cp = strstr(wp, "/*");
    /* Full comment line */
    if(cp == wp)
```

```

{
    /* If comment spans more than one line */
    while(strstr(buffer, "**/") == NULL)
    {
        comlines++;
        total++;
        fgets(buffer, 300, fp);
        if(ferror(fp))
        {
            printf("Error in reading file %s\n", fname);
            fclose(fp);
            free(buffer);
            return(ERROR);
        } /* if(!ferror) */
    }
    comlines++;
    continue;
}
/* Line of code */
else if(cp == NULL)
{
    codelines++;
    continue;
}
/* line of code with a comment */
else
{
    partlines++;
    codelines++;
}
}
printf("File: %s\n", fname);
printf(" code lines: %5d blank lines: %5d\n",
       codelines, blanklines);
printf(" comment lines: %5d partial lines: %5d\n",
       comlines, partlines);
printf(" total lines: %5d\n", total);
/* Clean up and return */
fclose(fp);
free(buffer);
return(COMPLETE);
}

```

Attributes

Async Safe	Yes
Cancel Safe	No

fopen()

Interrupt Handler Safe	No
------------------------	----

References

- ANSI X3.159-1989, §4.9.5.3
- **fclose()**, p. 190
- **fdopen()**, p. 193
- **freopen()**, p. 229
- **fs_init()**, p. 231

format()

be used if DOS is to access a hard disk. OS Open does not place any restrictions on placement of FAT file systems on hard disks.

—b Specifies number of blocks. If not specified, **format()** will try to determine number of blocks on the device.

—d Prints to **stdout** current partition table.

dev_name Specifies block device to be formatted with FAT file system.

If successful, **format()** returns 0. Otherwise, **format()** returns -1.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

Synopsis

```
#include <unistd.h>
long fpathconf( int fd, int name );
```

Library

devLib.a

Description

fpathconf() retrieves the value of a configurable limit or option specified by *name* and associated with the open file referenced by *fd*.

The configurable POSIX path name variables are:

_PC_LINK_MAX	Number of links per directory entry
_PC_MAX_CANON	Number of bytes in a terminal canonical input line (terminal special files only)
_PC_MAX_INPUT	Minimum number of bytes available in a terminal input queue (terminal special files only)
_PC_NAME_MAX	Maximum number of bytes in a file name
_PC_PATH_MAX	Maximum number of bytes in a path name
_PC_CHOWN_RESTRICTED	chown() is restricted to a thread with appropriate privileges
_PC_NO_TRUNC	Path name components longer than {NAME_MAX} generate an error
_PC_VDISABLE	Disable terminal special characters

If no limit is associated with the particular variable *name*, **fpathconf()** returns `-1` without altering *errno*. Otherwise, **fpathconf()** returns `-1` and sets *errno*.

Errors

[EBADF] *fd* does not refer to a valid file descriptor.

Example

See **mkdir()**, p. 427.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

fpathconf()

References

- IEEE Std 1003.1-1990, §5.7.1
- **pathconf()**, p. 486

Synopsis

```
#include <kadtLib.h>

int fpreg_dump( pthread_t thid, char *buf );
```

Library

kadtLib.a

Description

fpreg_dump() displays the floating-point registers from the thread identified by *thid*. The thread must have been created with the **PTHREAD_FP_AVAILABLE_NP** attribute for this function to work properly. This function will work in implementations that have real or emulated floating-point registers. Some implementations perform floating-point operations directly through the compiler function cells, and do not emulate floating-point registers.

If *buf* is NULL, the output is generated by **bprintf()**. Otherwise, the information is stored in the buffer pointed to by *buf*.

fpreg_dump() returns -1 if *thid* is not a valid thread ID, the thread was not created with the **PTHREAD_FP_AVAILABLE_NP** attribute or the implementation does not have floating-point registers. Otherwise, **fpreg_dump()** returns 0.

Note: This buffer should have a minimum capacity of 1100 characters.

Errors

None.

Example

This example shows **fpreg_dump()** called from **OpenShell**:

```
OS OPEN>fpreg_dump($dfthread)
-----
Floating-Point Registers from Thread 0x459dd8:
$fr0 0xbfb28bc65e78e742 $fr1 0x3fe5555555555555 $fr2 0x3fe3c9af78209765
$fr3 0x3de6064da2b39659 $fr4 0x3ec71de3a524f063 $fr5 0xbd6ad93cdc4b2455
$fr6 0xbf2a01a01a013e1a $fr7 0x400921fb54442d18 $fr8 0x0000000000000000
$fr9 0xbfc5555555555555 $fr10 0x3f8111111111110b0 $fr11 0xbd6ae420dc08499c
$fr12 0x0000000000000000 $fr13 0x0000000000000000 $fr14 0x0000000000000000
$fr15 0x0000000000000000 $fr16 0x0000000000000000 $fr17 0x0000000000000000
$fr18 0x0000000000000000 $fr19 0x0000000000000000 $fr20 0x0000000000000000
$fr21 0x0000000000000000 $fr22 0x0000000000000000 $fr23 0x0000000000000000
$fr24 0x0000000000000000 $fr25 0x0000000000000000 $fr26 0x0000000000000000
$fr27 0x0000000000000000 $fr28 0x0000000000000000 $fr29 0x0000000000000000
$fr30 0x0000000000000000 $fr31 0x0000000000000000 $fpscr 0x82068000
-----
OS OPEN>
```

fpreg_dump()

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84

Synopsis

```
#include <stdio.h>

int fprintf( FILE *stream,
             const char *format-string, ... );
```

Library

fsLib.a

Description

fprintf() formats and writes a series of characters and values to the output stream pointed to by *stream*.

fprintf() converts each entry in the argument list, if any, and writes to *stream* using the corresponding format specification in *format-string*, which has the same form and function as the *format-string* argument for **printf()**. See **printf()** on page 500 for a description of *format-string* and its argument list.

fprintf() cannot format and print more than 509 characters. If **fprintf()** tries to print more than 509 characters, results are unpredictable.

If successful, **fprintf()** function returns the number of characters printed. If an output error occurs, **fprintf()** returns a negative value.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **fread()**, p. 225.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.6.1
- **fprintf()**, p. 221
- **fs_init()**, p. 231
- **fscanf()**, p. 234
- **sprintf()**, p. 763

fputc()

Synopsis

```
#include <stdio.h>

int fputc( int c, FILE *stream );
```

Library

fsLib.a

Description

fputc() converts *c* to an unsigned character, writes the character to the stream pointed to by *stream*, and advances the file position indicator.

If an error occurs, **fputc()** returns **EOF**.

fputc() is equivalent to **putc()**, but **putc()** is a macro.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

The following example writes a string to a file until the end of the string or a new-line, and returns the number of characters written.

```
#include <stdio.h>

int write_str(FILE *file, char *string)
{
    int count = 0;
    char *wp = string;
    while((*wp != '\0') && (*wp != '\n'))
    {
        if(fputc(*wp++, file) == EOF)
        {
            count = 0;
            break;
        }
        count++;
    }
    return(count);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.7.3
- **fgetc()**, p. 198
- **fs_init()**, p. 231
- **putc()**, p. 609

fputs()

Synopsis

```
#include <stdio.h>

int fputs( const char *string, FILE *stream );
```

Library

fsLib.a

Description

fputs() writes the string pointed to by *string* into the stream pointed to by *stream*.

fputs() does not copy the NULL character (\0) at the end of the string.

If successful, **fputs()** returns a nonnegative value. Otherwise, **fputs()** returns **EOF**.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **fgetpos()**, p. 199.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.7.4
- **fgets()**, p. 201
- **fs_init()**, p. 231
- **gets()**, p. 289
- **puts()**, p. 615

Synopsis

```
#include <stdio.h>

size_t fread( void *buffer,
              size_t size,
              size_t count,
              FILE *stream );
```

Library

fsLib.a

Description

fread() reads *count* items of *size* length from the input stream pointed to by *stream*, stores them in the buffer pointed to by *buffer*, and advances the file position indicator by the number of bytes read.

fread() returns the number of complete items read, which can be less than *count* if end-of-file (EOF) is encountered or an error occurs before *count* items are read.

If either *size* or *count* is 0, **fread()** returns 0; the contents of *buffer* and the state of *stream* remain unchanged.

ferror() and **feof()** distinguish between a read error and EOF.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

The following example opens a binary file in read-only mode, reads 10 numbers from the file into an array, and writes the array to the standard output. The file is reopened in write-only mode and 10 random numbers are written to the file.

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

#define ERROR 1
#define COMPLETE 0

int number_switch(char *name)
{
    int num[10], i;
    FILE *file;
    /* Open the file in read only mode */
    file = fopen(name, "rb");
```

fread()

```
if(file == NULL)
{
    fprintf(stderr, "Error in opening file %s\n", name);
    return(ERROR);
}
/* Read in 10 numbers into the array */
if(fread((void *)&num[0], sizeof(int), 10, file) < 10)
{
    fprintf(stderr, "Error in reading from file %s\n", name);
    fclose(file);
    return(ERROR);
}
/* Print the numbers to stdout */
vprintf("%d, %d, %d, %d, %d,\n%d, %d, %d, %d\n",
        (va_list)&num[0]);
/* Reopen file in write only mode */
file = freopen(name, "wb", file);
if(file == NULL)
{
    fprintf(stderr, "Error in reopening file %s\n", name);
    return(ERROR);
}
/* Initialize array to random numbers */
srand(num[0]); /* set seed to first number in array */
for(i = 0; i < 10; i++)
    num[i] = rand();
/* Write the array of numbers to the file */
if(fwrite((void *)&num[0], sizeof(int), 10, file) < 10)
{
    fprintf(stderr, "Error in writing to file %s\n", name);
    fclose(file);
    return(ERROR);
}
/* Flush the file's buffer to write to the file */
if(fflush(file) == EOF)
{
    fprintf(stderr, "Error in flushing file's buffer\n");
    fclose(file);
    return(ERROR);
}
/* Close the file and return */
fclose(file);
return(COMPLETE);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.8.1
- **fread()**, p. 225
- **fs_init()**, p. 231
- **fwrite()**, p. 253

free()

Synopsis

```
#include <stdlib.h>
void free( void *ptr );
```

Library

cLib.a

Description

free() returns a block of storage to the heap.

ptr points to a block previously allocated by **calloc()**, **malloc()**, or **realloc()**. If *ptr* is NULL, **free()** simply returns.

free() returns nothing.

Note: Attempting to free a block of storage not allocated by **calloc()**, **malloc()**, or **realloc()** functions, or previously freed, may affect subsequent storage allocation and have undefined results.

Errors

None.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.3.2
- **calloc()**, p. 89
- **malloc()**, p. 398
- **realloc()**, p. 638

Synopsis

```
#include <stdio.h>

FILE *freopen( const char *filename,
               const char *type,
               FILE *stream );
```

Library

fsLib.a

Description

freopen() closes the file associated with the stream pointed to by *stream* and reassigns *stream* to the file name pointed to by *filename*.

freopen() can redirect the previously opened files **stdin**, **stdout**, and **stderr** to files that you specify. **freopen()** opens *filename* with *type*, a character string that specifies the type of access requested for *filename*.

See **fopen()** on page 210 for a list of permissible values of *type*.

If successful, **freopen()** returns the value of *stream* to *filename*. If an error occurs, the **freopen()** function closes the original file and returns a NULL pointer.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **fread()**, p. 225.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.5.4
- **fclose()**, p. 190
- **fopen()**, p. 210
- **fs_init()**, p. 231

frexp()

Synopsis

```
#include <math.h>

double frexp( double value, int *exp );
```

Library

mathLib.a

Description

frexp() breaks a floating-point number into a normalized fraction and an integral power of 2 stored in the object pointed to by *exp*.

frexp() returns the value *x*, such that *x* is a double with magnitude in the interval $[1/2, 1)$ or 0, and *value* equals *x* multiplied by 2 to the power pointed to by **exp*. If *value* is 0, both parts of the result are 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.4.2

Synopsis

```
#include <stdio.h>
int fs_init( void );
```

Library

fsLib.a

Description

fs_init() initializes the C Library file system.

Any C Library I/O function called before **fs_init()** will fail and return an error.

fs_init() requires three open file descriptors (0, 1, and 2) for **stdin**, **stdout**, and **stderr**, respectively. Before **fs_init()** is called, the low-level I/O system must be initialized and three **open()** calls must be made against an installed device driver to establish those descriptors.

The file system needs to be initialized only once. Subsequent calls to **fs_init()** are ignored.

If successful, **fs_init()** returns 0. Otherwise, **fs_init()** returns -1 and sets *errno*.

Errors

[ENOTSUP]	The low level I/O was not initialized.
[ENFILE]	The low level I/O did not initialize at least three open files.
[ENOMEM]	Insufficient memory is available to initialize the file system.

Example

The following example performs OS Open initialization, installing an async device driver, initializing the file system, setting the system clock, and changing the maximum number of semaphores allowed.

```
#include <sys/types.h>
#include <sys/devLib.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <sys/ksyscall.h>
#include <assert.h>
#include <unistd.h>
#include <stddef.h>
#include <stdlib.h>
```

```
#include <sys/asyncLib.h>
#include <iocclib.h>
#include <config.h>

int oldstate;
void main();
void panic();
const conf_t _Kernel_Config_block = {
    main,          /* Main function */
    panic,         /* Panic function */
    0x00000000,    /* Main argc */
    0x00040000,    /* Main stack size */
    (void *)0x00000000, /* Memory heap 1 start address */
    0x00500000,    /* Memory heap 1 size */
    0x00000000,    /* Memory heap 2 start address */
    0x00000000,    /* Memory heap 2 size */
    0x000004b0,    /* Trace table size */
    0x00fe5100,    /* Tick rate */
    0x0fe51000     /* Round robin time slice */
};

void main()
{
    int devhandle;
    int fd1, fd2, fd3;
    char stderr_buffer[80];
    int rv;
    /* Call Platform specific calls to initialize hardware */
    iocclnit();
    timertick_install();
    /* Change interruptibility state to disable */
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
    /* Initialize device driver subsystem */
    rv = dev_io_init(16, 16);
    if(rv < 0)
        abort();
    /* Identify async device driver to the device subsystem */
    rv = driver_install(&devhandle, async_init);
    if(rv < 0)
        abort();
    /* Associate async device driver with the name "/dev/tty0" */
    rv = device_install("/dev/tty0", CHRTYPE, devhandle, 1, 128, 128);
    if(rv < 0)
        abort();
    /* Open three file descriptors for stdin, stdout, and stderr */
    fd1 = open("/dev/tty0", O_RDONLY, asyncParityNone, asyncParityOdd,
```



```

        asyncStopBits1, asyncDataBits8, 9600);
fd2 = open("/dev/tty0", O_WRONLY, asyncParityNone, asyncParityOdd,
        asyncStopBits1, asyncDataBits8, 9600);
fd3 = open("/dev/tty0", O_WRONLY, asyncParityNone, asyncParityOdd,
        asyncStopBits1, asyncDataBits8, 9600);
if((fd1 < 0) || (fd2 < 0) || (fd3 < 0))
    abort();
/* Initialize the file system */
if(fs_init() < 0)
    abort();
/* Set stderr to no buffering */
setbuf(stderr, NULL);
/* Set the software clock */
clock_set(); /* Platform specific call */
/* Check to see if semaphores are supported with assert macro */
assert(sysconf(_SC_SEMAPHORES));
/* If this far, semaphores supported */
/* change maximum number of semaphores to 20 */
setsysconf(_SC_SEM_NSEMS_MAX, 20);
/* Set time zone for Central Standard time in the US */
/* Daylights savings time beginning on the first Saturday in May */
/* at 2:30 AM and ending on the fourth Saturday in October at */
/* 3:00 AM */
setenv_np("TZ=CST6CDT7,M5.1.6/02:30:00,M10.4.6/03:00:00");
/* set POSIX time zone global variable */
tzset();
/* enable interruptibility state */
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
        :
    }

```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **dev_io_init()**, p. 171
- **device_install()**, p. 172
- **driver_install()**, p. 177
- **open()**, p. 478

fscanf()

Synopsis

```
#include <stdio.h>

int fscanf( FILE *stream,
            const char *format-string, ... );
```

Library

fsLib.a

Description

fscanf() reads data from the stream pointed to by *stream* into locations given by entries in the argument list, if any.

fscanf() uses strings in the argument list to point to a variable of the type specified in *format-string* to interpret the input fields. *format-string* has the same form and function as the *format-string* argument for **scanf()**. See **scanf()** on page 681 for a description of *format-string*.

If successful, **fscanf()** returns the number of fields that it converted and assigned. This number does not include fields that **fscanf()** read but did not assign.

If an attempt to read at end-of-file (EOF) or an error occurs, **fscanf()** returns **EOF**.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **getenv()**, p. 262.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.6.2
- **fprintf()**, p. 221
- **fs_init()**, p. 231
- **scanf()**, p. 681

Synopsis

```
#include <stdio.h>

int fseek( FILE *stream,
           long int offset,
           int origin );
```

Library

fsLib.a

Description

fseek() sets the file position indicator of the input stream pointed to by *stream* to a location *offset* bytes from *origin*.

The next operation on *stream* takes place at the new location. If *stream* is open for update, the next operation can be a read or a write.

origin must be one of the following constants defined in the file **<stdio.h>**:

SEEK_SET	Beginning of file
SEEK_CUR	Current file position indicator
SEEK_END	End of file.

An attempt to set the position before the beginning of the file causes an error.

fseek() clears the end-of-file (EOF) indicator, even when *origin* is **SEEK_END**, and undoes any preceding **ungetc()** function call on *stream*.

If successful, **fseek()** returns 0. Otherwise, **fseek()** returns a nonzero value. On devices that cannot seek, such as terminals and printers, the return value is nonzero.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **fgetpos()**, p. 199.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

fseek()

References

- ANSI X3.159-1989, §4.9.9.2
- **fs_init()**, p. 231
- **ftell()**, p. 239
- **rewind()**, p. 652

Synopsis

```
#include <stdio.h>

int fsetpos( FILE *stream,
             const fpos_t *pos );
```

Library

fsLib.a

Description

fsetpos() sets the file position indicator of the stream pointed to by *stream* to the location, pointed to by *pos*, obtained by a previous call to **fgetpos()**.

fsetpos() function always clears the end-of-file (EOF) indicator and undoes any previous **ungetc()** function call on *stream*.

If successful, **fsetpos()** returns 0. Otherwise, **fsetpos()** returns a nonzero value.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **fgetpos()**, p. 199.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.9.3
- **fgetpos()**, p. 199
- **fs_init()**, p. 231
- **ftell()**, p. 239
- **rewind()**, p. 652

fstat()

Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
int fstat( int fd, struct stat *buf );
```

Library

devLib.a

Description

fstat() stores information about the open file referenced by *fd* in the object pointed to by *buf*.

If successful, **fstat()** returns 0. Otherwise, **fstat()** returns -1 and sets *errno*.

Errors

[EBADF] *fd* does not refer to a valid file descriptor.

Example

See **access()**, p. 55.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.6.2
- See **stat()**, p. 770.

Synopsis

```
#include <stdio.h>
long int ftell( FILE *stream );
```

Library

fsLib.a

Description

ftell() returns the value of the file position indicator associated with the stream pointed to by *stream*.

For a fixed-length binary file, **ftell()** returns an offset relative to the beginning of *stream()*.

If an error occurs, **ftell()** returns EOF.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

The following example returns the number of bytes currently in the file.

```
#include <stdio.h>

long int how_big(FILE *file)
{
    long int num_bytes, p;
    /* Save current position in file */
    p = ftell(file);
    /* set file position to end of file */
    fseek(file, 0, SEEK_END);
    /* file position is the number of bytes in the file */
    num_bytes = ftell(file);
    /* Restore file back to original position */
    fseek(file, p, SEEK_SET);
    return(num_bytes);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	Yes

ftell()

References

- ANSI X3.159-1989, §4.9.9.4
- **fgetpos()**, p. 199
- **fopen()**, p. 210
- **fs_init()**, p. 231
- **fseek()**, p. 235

Synopsis

```
#include <ftp.h>
int ftp( char *args );
```

Library

ftpLib.a

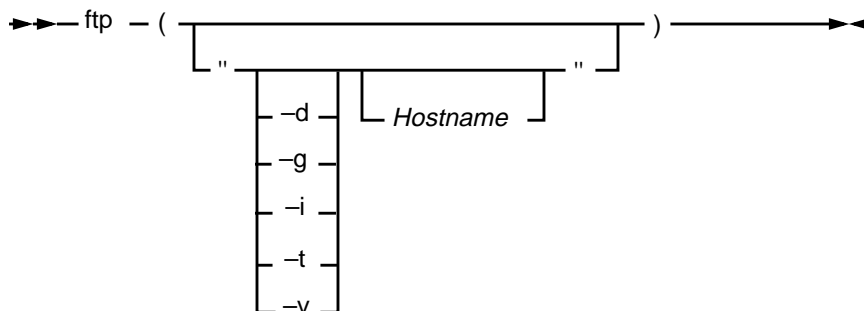
Description

ftp() provides an implementation of the TCP/IP File Transfer Protocol (FTP) client that supports file transfers between the local system and a remote system.

The remote system, called a “remote host,” must be running an `ftpd` daemon.

Typically, an **ftp** command is entered on a shell command line and runs a memory-resident **ftp()** function implemented using the synopsis prototype. **ftp()** provides the local FTP user interface. *args* points to command arguments.

The following diagram illustrates the command syntax.



ftp accepts the following flags:

- `-d` Print debugging information
 - `-g` Disable “globbing” (file name expansion using metacharacters)
 - `-i` Turn off prompting during multiple file transfers
 - `-t` Turn on tracing
 - `-v` Verbose; provide responses from the remote host and file transfer statistics.
- If **ftp** sends the standard output to a display, `-v` is in effect by default. If output is redirected to a file, verbose mode is in effect only if `-v` is specified.

Hostname specifies the machine with which files are exchanged.

debug	If on, prints each command sent to the remote host, preceded by the string <code>-></code> . Toggles on and off.
delete <i>RemoteFile</i>	Erases the specified file on the remote host.
dir [<i>RemoteDirectory</i>]	Prints a detailed listing of the files in the specified directory on the remote host. By default, lists the working directory.
disconnect	Synonym for the close subcommand.
get <i>RemoteFile</i> [<i>LocalFile</i>]	Copies <i>RemoteFile</i> from the remote host to the local host. If <i>LocalFile</i> is not given, the remote file name is used.
glob	<p>Toggles globbing (file name expansion using metacharacters) for the mdelete, mget and mput subcommands. If the -g flag is not set, file name parameters for these subcommands are not expanded. If globbing is on and a pattern-matching character is used in a subcommand that expects one file name, results may be unpredictable.</p> <p>For example, the append and put subcommands perform file name expansion and then use only the first file name generated. Other ftp subcommands, such as cd, delete, get, mkdir, rename, and rmdir do not expand file names expansion and take the metacharacters literally.</p> <p>Globbing for the mput subcommand is done locally. For the mdelete and mget subcommands, each file name is expanded separately at the remote machine and the lists are not merged. Expansion of a directory name may differ from expansion of a filename, depending on the remote host and its ftpd server.</p> <p>To preview the expansion of a directory name, use the mls subcommand: mls remotefile</p>
hash	Toggles hash mark (#) printing. When the hash subcommand is on, ftp displays a hash mark for each transferred 1 KB data block.
help [<i>Subcommand</i>]	Displays help information for <i>Subcommand</i> .
idle	Synonym for the site idle subcommand.
image	Synonym for the binary subcommand.
lcd [<i>Directory</i>]	Changes the working directory on the local host.

ls [<i>RemoteDirectory</i>]	Lists files on the remote host. <i>RemoteDirectory</i> can name a directory or file, and can contain wildcards such as <code>"*"</code> .
macdef <i>Macro</i>	<p>Defines a subcommand macro. Subsequent lines until a null line (two consecutive line-feeds) are saved as the text of the macro. Up to 16 macros, containing at most 4096 characters, can be defined. Macros remain defined until either redefined or a close or disconnect subcommand is executed.</p> <p>The \$ (dollar sign) and \ (backslash) are special characters in ftp macros.</p> <p>A \$ followed by a number is replaced by the corresponding macro parameter at invocation. A \$ followed by a letter i indicates the macro is to loop; the \$i is replaced by consecutive parameters on each pass. (See the \$ subcommand.)</p> <p>A \ symbol escapes the following character. Use \ symbol to turn off the special meaning of the \$ and \ symbols.</p>
mdelete <i>RemoteFiles</i>	Expands the files specified by <i>RemoteFiles</i> at the remote host and deletes the remote files.
mdir [<i>RemoteDirectories LocalFile</i>]	<p>Expands directories specified by <i>RemoteDirectories</i> at the remote host and writes a list of the contents of the directories to the file specified by <i>LocalFile</i>. If the <i>RemoteDirectories</i> parameter contains a pattern-matching character and no local file is given, the mdir subcommand prompts for a local file name. If the <i>RemoteDirectories</i> parameter is a list of remote directories separated by blanks, the last argument in the list must be a local file or a - (hyphen).</p> <p>If the <i>LocalFile</i> parameter is - (hyphen), the mdir subcommand writes the list to standard output. If interactive prompting is on (see the prompt subcommand), ftp prompts to confirm that the last parameter is a local file, not a remote directory.</p>
mget <i>RemoteFiles</i>	Copies the <i>RemoteFiles</i> from the remote host to the local host. The <i>RemoteFiles</i> parameter can contain wildcards such as <code>"*"</code> .
mkdir [<i>RemoteDirectory</i>]	Creates the specified directory on the remote host.

mls [*RemoteDirectories* *LocalFile*]

Expands the directories specified by *RemoteDirectories* at the remote host and writes an abbreviated list of the contents of the directories to the *LocalFile*. If the *RemoteDirectories* parameter contains a pattern-matching character and no local file name is given, **mls** prompts for a local file name. If the *RemoteDirectories* parameter is a list of remote directories separated by blanks, the last argument in the list must be a local filename or a - (hyphen).

If the *LocalFile* parameter is - (hyphen), the **mls** subcommand writes the list to the standard output. If interactive prompting is on (see the **prompt** subcommand), **ftp** prompts to confirm that the last parameter is a local file, not a remote directory.

modtime *RemoteFile*

Shows the modification time of *RemoteFile* on the remote host.

mput *LocalFiles*

Copies *LocalFiles* on the local host to the remote host. The *LocalFiles* parameter can contain wildcards such as `"**"`.

newer [*RemoteFile*] [*LocalFile*]

Gets *RemoteFile* if it is newer than *LocalFile*. If not, a message is displayed and the file is not transferred.

nlist [*RemoteDirectory*] [*LocalFile*]

Synonym for the **ls** subcommand.

open *Hostname*

Establishes a connection to the **ftpd** daemon on the named remote host.

prompt

Toggles interactive prompting for verification before sending or receiving multiple files with the **mdelete**, **mget**, or **mput** subcommands.

put *LocalFile* [*RemoteFile*]

Copies *LocalFile* from the local host to the remote host. If *RemoteFile* is not given, the local file name is used.

pwd

Prints the current directory on the remote host.

quit

Closes the connection and exits **ftp**.

recv *RemoteFile* [*LocalFile*]

Synonym for the **get** subcommand.

rename *FromName ToName*

Renames a file on the remote host.

rhelp [*Subcommand*]

Requests help from the remote host.

rmdir *RemoteDirectory*

Removes the specified directory on the remote host.

rstatus

Shows the status of the remote host.

runique

(Receive unique). Toggles unique file name creation for local destination files during **get** and **mget** subcommands. If **runique** is off (the default), **ftp** may overwrite local files. Otherwise, if a local file has the same name as a local destination file, **ftp** appends “.1” to the name of the local destination file.

If a local file already has the new name, **ftp** appends “.2” to the local destination file name. If a local file has this name, **ftp** continues incrementing the postfix until it finds a unique name or reaches “.99” without finding a unique name. If **ftp** cannot find a unique name, it reports an error and stops the transfer.

send *LocalFile* [*RemoteFile*]

Synonym for the **put** subcommand.

site *Args*

If *Args* is **idle**, this subcommand displays or sets the idle time-out period on the remote host.

If *Args* is **umask**, this subcommand sets the file creation umask on the remote host.

If *Args* is **chmod**, this subcommand uses **chmod** change the permissions of a file on the remote host.

status

Displays the status of the **ftp** session.

sunique

(Send/store unique). Toggles unique file name creation for remote destination files during **put** and **mput** subcommands. If **sunique** is off (the default), **ftp** overwrites remote files. Otherwise, if a remote file has the same name as a remote destination file, the remote **ftpd** server modifies the remote destination file name. Note that the remote server must support the STOU instruction.

system

Shows the operating system of the remote host.

trace

Toggles packet tracing.

type [ascii binary]	Sets the file transfer type. <i>ascii</i> sets the file transfer type to network ASCII and is the default. <i>binary</i> should be used for transferring binary images.
umask	Synonym for site umask subcommand.
user User [Password]	Logs in <i>User</i> on the remote host. If the remote host requires a password and none is given, ftp prompts for one.
verbose	Toggles verbose mode. If verbose is on (the default), ftp displays responses from the remote host and file transfer statistics.

Example

The following example shows **ftp** being run from a shell prompt. User commands appear in bold type.

The user logs in as “elvis” on the remote system named “graceland.” The password is not echoed.

The local directory is changed to **/fat**, which may be a file system that was previously installed using **device_install()**.

The remote directory is changed to **hits**, an **ls** subcommand lists its contents, and a **get** subcommand copies the file **jailhouse** from the remote host to the working directory of the local OS Open system. A **bye** subcommand ends the session.

```
OS Open> ftp("graceland")
Connected to graceland
220 village FTP Server (Version number 2)
Name (village): elvis
331 Password required for elvis
Password:
230 User elvis logged in
ftp> lcd /fat
Local directory now /fat
ftp> cd hits
250 CWD command successful
ftp> ls
200 PORT command successful
150 Opening data connection for /bin/ls
bluesuede
jailhouse
loveme
226 Transfer complete
ftp> get jailhouse
200 PORT command successful
150 Opening data connection for jailhouse (1977 bytes)
```

ftp()

```
226 Transfer complete
1977 bytes received in 3 seconds (0.643 Kbytes/s)
ftp> bye
221 Goodbye - be seeing you
OS Open>
```

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- `ftpd_start()`, p. 249

Synopsis

```
#include <ftp.h>
void ftpd_start( void * args )
```

Library

ftpLib.a

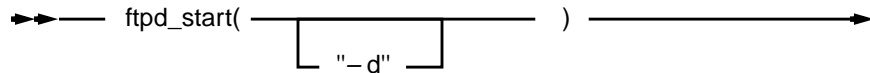
Description

ftpd_start() starts the ftpd daemon thread and returns to the caller.

The ftpd daemon thread continues to run in the background

To start a connection to the OS Open ftpd server from a remote client, such as an AIX system, the caller must provide an OS Open user name. Because OS Open does not validate users, typing any user ID or pressing the Enter key at the client *Name:* prompt is sufficient.

The following syntax diagram illustrates the syntax of the **ftpd_start** command, which is entered from a command line:



The **-d** flag causes **ftpd_start** to print debugging information.

Errors

None.

Example

In the following example, the **ftpd** daemon is started in its own thread. This call may be made from a program or from OpenShell. Control returns to the caller immediately. The thread running the **ftpd** daemon continues to run in the background, ready to service incoming connections from remote hosts.

```
ftpd_start();
```

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- **ftp()**, p. 241

ftruncate()

Synopsis

```
#include <unistd.h>
int ftruncate( int fildes, off_t length );
```

Library

devLib.a

Description

ftruncate() sets the size of the file *fildes* to *length*.

The file must be open for writing, and must be a regular file or a shared memory object. If the file was originally larger than *length*, the extra data is discarded. If the file was originally smaller, its size is increased.

If successful, **ftruncate()** returns 0. Otherwise, **ftruncate()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>fildes</i> does not refer to a valid file descriptor.
[EINVAL]	<i>fildes</i> refers to a file for which this operation is not valid.
[EROFS]	<i>fildes</i> refers to a file that is not open for writing.

Example

See **shm_open()**, p. 734.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4/D13 §5.6.7

Synopsis

```
#include <stdio.h>
int ftrylockfile( FILE *stream );
```

Library

fsLib.a

Description

ftrylockfile() is the non-blocking version of **flockfile()**.

If successful, **ftrylockfile()** returns 0. Otherwise, **ftrylockfile()** returns a nonzero value to indicate that the lock cannot be acquired.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8 §19.2.2.1
- **fs_init()**, p. 231
- **funlockfile()**, p. 252

funlockfile()

Synopsis

```
#include <stdio.h>
void funlockfile( FILE *stream );
```

Library

fsLib.a

Description

funlockfile() relinquishes exclusive access to the stream pointed to by *stream*.

A thread may have nested matching calls to **flockfile()** and **funlockfile()** functions for a single stream. In such a case, only the first call to **flockfile()** and the matching call to **funlockfile()** have any effect.

funlockfile() returns nothing.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **flockfile()**, p. 205.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §19.2.2.1
- **flockfile()**, p. 205
- **fs_init()**, p. 231

Synopsis

```
#include <stdio.h>

size_t fwrite( const void *buffer,
               size_t size,
               size_t count,
               FILE *stream );
```

Library

fsLib.a

Description

fwrite() writes *count* items of *size* bytes from the buffer pointed to by *buffer* to the output stream pointed to by *stream*.

If successful, **fwrite()** returns the number of items written, which can be fewer than *count* if an error occurs.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **fread()**, p. 225.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.8.2
- **fopen()**, p. 210
- **fread()**, p. 225
- **fs_init()**, p. 231

get_myaddress()

Synopsis

```
#include <netinet/in.h>
#include <rpc/rpc.h>

int get_myaddress( struct sockaddr_in *addr );
```

Library

rpcLib.a

Description

get_myaddress() gets a machine's IP address without using library routines that access the **_Tcpiip_hosts** array.

addr points to the machine's IP address. The port number is set to a value of **htons(PMAPPORT)**.

If successful, **get_myaddress()** returns 0. Otherwise, **get_myaddress()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <stdio.h>

int getc( FILE *stream );
```

Library

None. This is a macro defined in `<stdio.h>`.

Description

getc() returns the next character from the input stream pointed to by *stream* and increases the file position indicator.

getc() is equivalent to **fgetc()**, except that **getc()** is a macro and arguments may be evaluated more than once. *stream* should not be an expression with side effects.

If **getc()** reads the end-of-file (EOF), or if an error occurs, **getc()** returns EOF. Use **ferror()** or **feof()** to determine whether an error or an EOF condition occurred.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **ungetc()**, p. 921.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.7.5
- **fgetc()**, p. 198
- **fs_init()**, p. 231
- **gets()**, p. 289
- **puts()**, p. 615
- **ungetc()**, p. 921

getc_unlocked()

Synopsis

```
#include <stdio.h>
int getc_unlocked( FILE *stream );
```

Library

fsLib.a

Description

getc_unlocked() is equivalent to **getc()**, except that **getc_unlocked()** is not async safe.

getc_unlocked() can be safely used only within code protected by **flockfile()** and **funlockfile()** calls.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **flockfile()**, p. 205.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §19.2.2.2
- **fs_init()**, p. 231
- **getc()**, p. 255

Synopsis

```
#include <stdio.h>

int getchar( void );
```

Library

None. This is a macro defined in `<stdio.h>`.

Description

getchar() reads a character from the standard input at the file position indicator, returns the character read, and advances the file position indicator to the next character.

If **getchar()** reads from end-of-file (EOF), or if an error occurs, **getchar()** returns EOF. Use **ferror()** or **feof()** to determine whether an error or an EOF condition occurred.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **scanf()**, p. 681.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.7.6
- **fgetc()**, p. 198
- **fs_init()**, p. 231
- **gets()**, p. 289
- **putchar()**, p. 612
- **ungetc()**, p. 921

getchar_unlocked()

Synopsis

```
#include <stdio.h>
int getchar_unlocked( void );
```

Library

None. This is a macro defined in **<stdio.h>**.

Description

getchar_unlocked() is equivalent to **getchar()** except that **getchar_unlocked()** is not async safe.

getchar_unlocked() can be safely only used within code protected by **flockfile()** and **funlockfile()** calls.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

The following example reads commands from the standard input. The code illustrates two specific commands, exit and append. To see how the example works, run the example function and enter a few append commands, each followed by a character string. Enter an exit command. You should see that the entered strings were appended in sequence to a string.

```
#include <sys/types.h>
#include <stddef.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <pthread.h>
#include <limits.h>
#include <memLib.h>
#include <stdlib.h>

void cmd(void)
{
    char cmd_buf[120];
    char *wp;
    char *exit_cmd = {"exit"};
    char *append_cmd = {"append"};
    char *outbuf;
```

```
outbuf = (char *)malloc 256;
outbuf[0] = '\0';
/* Loop forever */
while(1)
{
    /* Get the command one byte at a time */
    wp = cmd_buf;
    flockfile(stdin);
    flockfile(stdout);
    while((*wp = getchar_unlocked()) != '\n')
    {
        putchar_unlocked(*wp);
        wp++;
    }
    funlockfile(stdin);
    funlockfile(stdout);
    /* Remove new line character and terminate string */
    *wp = '\0';
    /* Command is case insensitive, change to lowercase */
    wp = cmd_buf;
    while(*wp != '\0')
    {
        *wp = tolower(*wp);
        wp++;
    }
    /* Check for the exit command */
    if(strcmp(cmd_buf, exit_cmd) == 0)
    {
        /* Print the final results */
        raise(SIGTERM);
        pthread_exit(&addr);
    }
    :
    •
    /* Check for the append command */
    if(strncmp(cmd_buf, append_cmd, strlen(append_cmd)) == 0)
    {
        /* get the address entered */
        wp = cmd_buf + strlen(extend_cmd);
        /* ignore any white space between command and address */
        while(isspace(*wp))
            wp++;
        /* append the input to the output buffer */
        strcat(out_buf, wp);
    }
    •
}
```

getchar_unlocked()

```
        •  
    }  
}
```

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §19.2.2.2
- **fs_init()**, p. 231
- **getchar()**, p. 257

Synopsis

```
#include <unistd.h>
char *getcwd( char *buf, size_t size );
```

Library

devLib.a

Description

getcwd() copies the absolute path name of the working directory to the array of characters pointed to by *buf*.

size is the size in bytes of the array *buf* points to.

If *buf* is NULL, the behavior of **getcwd()** is undefined.

If successful, **getcwd()** returns *buf*. Otherwise, **getcwd()** returns a NULL pointer and sets *errno*; the contents of *buf* are undefined.

Errors

[EINVAL]	<i>size</i> is 0.
[ERANGE]	<i>size</i> is greater than 0 but smaller than the length of the path name plus 1.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, §5.2.2

getenv()

Synopsis

```
#include <stdlib.h>

char *getenv( const char *varname );
```

Library

cLib.a

Description

getenv() searches the environment variable table and returns a pointer to the environment table entry containing the value of *varname*.

If *varname* is not defined, **getenv()** returns a NULL value.

Errors

None.

Example

The following example reads an environment variable from **stdin** and writes the variable's definition to **stdout**. If the variable is not found, **getenv()** writes an error message.

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

void prt_ev(void)
{
    char buffer[80];
    char *is;

    printf("Enter an environment variable:\n");
    fscanf(stdin, "%s", buffer);
    if((is = getenv(buffer)) == (char *)NULL)
        printf("Can't find %s\n", buffer);
    else
        printf("%s = %s\n", buffer, is);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.4.4

Synopsis

```
#include <netdb.h>

struct hostent *gethostbyaddr( char *address, int len, int type );
```

Library

netLib.a

Description

gethostbyaddr() retrieves host information using a host address as a search key.

gethostbyaddr() recognizes domain name servers as described in RFC883. If the **_Tcpip_resolv** database contains a name server entry, **gethostbyaddr()** queries the domain name server. If the request to the domain name server times out, **gethostbyaddr()** checks the local **_Tcpip_hosts** database.

gethostbyaddr() returns a pointer to a **hostent** structure containing information from a domain name server or a field from an entry in the **_Tcpip_hosts** database. The **hostent** structure is defined in the file **<netdb.h>**.

address points to a host address passed as a hexadecimal string formatted for the address family of domain *type*. For example, the **AF_INET** address family uses a dotted decimal Internet Protocol (IP) format. *len* specifies the length of host address. **gethostbyaddr()** currently supports only the **AF_INET** address family; **AF_INET** is defined in **<sys/socket.h>**.

If an error occurs or the end of the database is reached, **gethostbyaddr()** returns a NULL pointer.

Note: The return value points to static data that subsequent calls to host name retrieval overwrite. Use **gethostlock()** and **gethostunlock()** to guard the **_Tcpip_hosts** database and static return value.

Errors

None.

Example

See **gethostunlock()**, p. 268.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

gethostbyaddr()

References

- `gethostbyname()`, p. 265
- `gethostlock()`, p. 266
- `gethostunlock()`, p. 268

Synopsis

```
#include <netdb.h>

struct hostent *gethostbyname( char *name );
```

Library

netLib.a

Description

gethostbyname() retrieves host information using a host name as a search key.

gethostbyname() recognizes domain name servers as described in RFC883. If the **_Tcpiplib_resolv** database contains a name server entry, **gethostbyname()** queries the domain name server. If the request to the domain name server times out, **gethostbyname()** checks the local **_Tcpiplib_hosts** database.

gethostbyname() returns a pointer to a **hostent** structure containing information from a name server program or a field from an entry in the **_Tcpiplib_hosts** database. The **hostent** structure is defined in the file **<netdb.h>**. *name* points to the host name to be found.

If an error occurs or the end of the database is reached, the **gethostbyname()** function returns a NULL pointer.

Note: The return value points to static data that subsequent calls to host name retrieval overwrite. Use **gethostlock()** and **gethostunlock()** to guard the **_Tcpiplib_hosts** database and static return value.

Errors

None.

Example

See **gethostunlock()**, p. 268.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- **gethostbyaddr()**, p. 263
- **gethostlock()**, p. 266
- **gethostunlock()**, p. 268

gethostlock()

Synopsis

```
#include <sys/netLib.h>
int gethostlock( void );
```

Library

netLib.a

Description

gethostlock() locks (acquires exclusive use of) the **_Tcpip_hosts** database.

Before any call to **gethostbyaddr()** and **gethostbyname()** is made **gethostlock()** function should be called.

If successful, **gethostlock()** returns 0. Otherwise, **gethostlock()** returns -1.

Errors

None.

Example

See **gethostunlock()**, p. 268.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **gethostbyaddr()**, p. 263
- **gethostbyname()**, p. 265
- **gethostunlock()**, p. 268

Synopsis

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

int gethostname( char *name, int namelen );
```

Library

netLib.a

Description

gethostname() returns the standard host name of the local host in the string pointed to by *name*.

If extra space is provided, *name* is null-terminated. If insufficient space is provided, *name* is truncated to fit the available space. *namelen* specifies the length of the *name* array.

System host names are limited to **MAXHOSTNAMELEN**, as defined in the file **<sys/netLib.h>**.

If successful, **gethostname()** returns 0. Otherwise, **gethostname()** returns **-1** and sets *errno*.

Errors

[EFAULT] *name* is NULL or *namelen* is set to 0.

Example

See **gethostunlock()**, p. 268.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **gethostbyaddr()**, p. 263
- **gethostbyname()**, p. 265

gethostunlock()

Synopsis

```
#include <sys/netLib.h>
void gethostunlock( void );
```

Library

netLib.a

Description

gethostunlock() unlocks (releases exclusive use of) the **_Tcpip_hosts** database.

gethostunlock() returns nothing.

Errors

None.

Example

The following example prints the local host name and searches for a remote host name by name and address.

```
#include <sys/netLib.h>
#include <sys/socket.h>

char          name[MAXHOSTNAMELEN];
struct hostent *h;
struct sockaddr_in sin;

        :

if (gethostname(name, sizeof(name))) {
    (void)printf("gethostname() failed\n");
    return(-1);
}
(void)printf("local host name %s\n", name);
if (gethostlock()) {
    (void)printf("gethostlock() failed\n");
    return(-1);
}
h=gethostbyaddr((char *)&sin->sin_addr, sizeof(struct in_addr), AF_INET);
if (h!=NULL) {
    (void)printf("host one name is %s\n", h->h_name);
}
h=gethostbyname("tanis.raleigh.ibm.com");
if (h!=NULL) {
    (void)printf("host two name is %s\n", h->h_name);
}
(void)gethostunlock()
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **gethostbyaddr()**, p. 263
- **gethostbyname()**, p. 265
- **gethostlock()**, p. 266

getnetbyaddr()

Synopsis

```
#include <netdb.h>
#include <sys/socket.h>

struct netent *getnetbyaddr( long net, int type );
```

Library

netLib.a

Description

getnetbyaddr() retrieves an entry from the **_Tcpip_networks** database, using a network address as a search key.

getnetbyaddr() searches the database sequentially from its start until the function encounters a matching net number and type or reaches the end of the database.

net specifies the network number to be located. *type* specifies the address family for the network; its only supported value is **AF_INET**, which is defined in **<sys/socket.h>**.

getnetbyaddr() returns a pointer to a **netent** structure that contains the contents of a network description entry in the **_Tcpip_networks** database. The **netent** structure is defined in the file **netdb.h**.

If an error occurs or the end of the database is reached, **getnetbyaddr()** returns a NULL pointer.

Note: The return value points to static data that subsequent calls overwrite. Use **getnetlock()** and **getnetunlock()** to guard the **_Tcpip_networks** database and static return value.

Errors

None.

Example

The following example prints a network name.

```
struct netent *network;
:
:
```

```
if (getnetlock()) {
    (void)printf("getnetlock() failed\n");
    return(-1);
}
network=getnetbyaddr(0, AF_INET);
if (network!=NULL) {
    (void)printf("network name is %s\n", network->n_name);
}
(void)getnetunlock();
```

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getnetbyname()**, p. 272
- **getnetlock()**, p. 273
- **getnetunlock()**, p. 274

getnetbyname()

Synopsis

```
#include <netdb.h>

struct netent *getnetbyname( const char *name );
```

Library

netLib.a

Description

getnetbyname() retrieves an entry from the **_Tcpiip_networks** database, using a network name as a search key.

getnetbyname() searches the **_Tcpiip_networks** database sequentially from its start until the function finds a matching net name or reaches the end of the database.

name points to a string containing the network name to be located.

getnetbyname() returns a pointer to a **netent** structure that contains the contents of a network description entry in the **_Tcpiip_networks** database. The **netent** structure is defined in the file **netdb.h**.

If an error occurs or the end of the database is reached, **getnetbyname()** returns a NULL pointer.

Note: The return value points to static data that subsequent calls overwrite. Use **getnetlock()** and **getnetunlock()** to guard the **_Tcpiip_networks** database and the static return value.

Errors

None.

Example

See **inet_network()**, p. 314.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getnetbyaddr()**, p. 270
- **getnetlock()**, p. 273
- **getnetunlock()**, p. 274

Synopsis

```
#include <sys/netLib.h>
int getnetlock( void );
```

Library

netLib.a

Description

getnetlock() locks (acquires exclusive use of) the **_Tcpip_networks** database.

getnetlock() should be called before making calls to **getnetbyaddr()** and **getnetbyname()**.

If successful, **getnetlock()** returns 0, Otherwise, **getnetlock()** returns -1.

Errors

None.

Example

See **inet_network()**, p. 314.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **getnetbyaddr()**, p. 270
- **getnetbyname()**, p. 272
- **getnetunlock()**, p. 274

getnetunlock()

Synopsis

```
#include <sys/netLib.h>
void getnetunlock( void );
```

Library

netLib.a

Description

getnetunlock() unlocks (releases exclusive use of) the **_Tcpip_networks** database.

getnetunlock() returns nothing.

Errors

None.

Example

See **inet_network()**, p. 314.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getnetbyaddr()**, p. 270
- **getnetbyname()**, p. 272
- **getnetlock()**, p. 273

Synopsis

```
int getopt( int nargc, const * char *nargv, const *ostr );
```

Library

runLib.a

Description

getopt() is used by programs which are invoked by **run()**. **getopt()** returns the next flag letter in the *nargv* parameter list that matches a letter in the *ostr* parameter. **getopt()** is an aid to help programs interpret command lines passed by the **run()** function.

The *optarg* external variable is set to point to the start of the flag's parameter on return from **getopt()**. **getopt()** places the *nargv* index of the next argument to be processed in *optind*. *optind* is set to 1 by **run()** so that *nargv[0]* is not processed.

When all flags have been processed, **getopt()** returns EOF. If no errors are detected, **getopt()** returns the current index of *ostr*. Otherwise it returns the ASCII value of '?' (63).

Errors

None.

Example

The following example show a program looking for the -n and -v flags.

```
int main(int argc, char **argv) {
    extern int optind;
    int ch;

    while ((ch = getopt(argc, argv, "nv")) != EOF)
        switch (ch) {
            case 'n':
                nflag = TRUE;
                break;
            case 'v':
                vflag = TRUE;
                break;
            case '?':
                (void)fprintf(stderr,
                    "usage: %s [-nv] file \n", argv[0]);
                return -1;
        }
    argv += optind;
    .
    .
    .
}
```

getopt()

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- `run()`, p. 679

Synopsis

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

int getpeername( int s, struct sockaddr *name, int *namelen );
```

Library

tcpiplib.a

Description

getpeername() returns the name of the peer connected to socket *s*.

namelen should initially indicate the amount of space pointed to by *name*. On return, *namelen* contains the size of the *name* that was returned. *name* is truncated if the buffer provided is too small.

getpeername() operates only on connected sockets.

If successful, **getpeername()** returns 0. Otherwise, **getpeername()** returns -1 and sets *errno*.

Errors

[EBADF]	The descriptor <i>s</i> is not valid.
[ENOTSOCK]	The descriptor <i>s</i> references a file, not a socket.
[ENOTCONN]	<i>s</i> is not connected.
[ENOBUFS]	Insufficient resources.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.6
- **accept()**, p. 52
- **bind()**, p. 80
- **getsockname()**, p. 295
- **socket()**, p. 760

getpid()

Synopsis

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

pid_t getpid( void );
```

Library

rtxLib.a

Description

getpid() returns the process ID of the calling process.

getpid() always succeeds, and is provided for POSIX compatibility.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, §4.1
- **kill()**, p. 350

Synopsis

```
#include <shell.h>
char *getprompt();
```

Library

shell.o

Description

getprompt() returns the location of the OpenShell prompt string.
The prompt string is a NULL-terminated string containing no more than 15 characters.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

getprotobyname()

Synopsis

```
#include <netdb.h>

struct protoent *getprotobyname( const char *name );
```

Library

netLib.a

Description

getprotobyname() retrieves an entry from the **_Tcpi_protocols** database, using a protocol name as a search key.

getprotobyname() can find a protocol's name, aliases, and protocol number.

getprotobyname() searches the **_Tcpi_protocols** database sequentially from its start until the function finds a matching protocol name or reaches the end of the database.

name specifies the protocol name to search by.

getprotobyname() returns a pointer to a **protoent** structure that contains the contents of an entry in the **_Tcpi_protocols** database. The file **netdb.h** defines the **protoent** structure.

If an error occurs or the end of the database is reached, **getprotobyname()** returns a NULL pointer.

Note: The return value points to static data that subsequent calls that retrieve protocols overwrite. Use **getprotolock()** and **getprotounlock()** to guard the **_Tcpi_protocols** database and static return value.

Errors

None.

Example

See **getprotounlock()**, p. 283.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getprotobyname()**, p. 281
- **getprotolock()**, p. 282
- **getprotounlock()**, p. 283

Synopsis

```
#include <netdb.h>

struct protoent *getprotobynumber( int protocol );
```

Library

netLib.a

Description

getprotobynumber() retrieves an entry from the **_Tcpi_protocols** database, using a protocol number as a search key.

getprotobynumber() can find a protocol's name, aliases, and protocol number.

getprotobynumber() searches the **_Tcpi_protocols** database sequentially from its start until the function finds a matching protocol number or reaches the end of the database.

protocol specifies the protocol number to search by.

getprotobynumber() returns a pointer to a **protoent** structure that contains contents of an entry in the **_Tcpi_protocols** database. The file **netdb.h** defines the **protoent** structure.

If an error occurs or the end of the database is reached, **getprotobynumber()** returns a NULL pointer

Note: The return value points to static data that subsequent calls that retrieve protocols overwrite. Use **getprotolock()** and **getprotounlock()** to guard the **_Tcpi_protocols** database and static return value.

Errors

None.

Example

See **getprotounlock()**, p. 283.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getprotobyname()**, p. 280
- **getprotolock()**, p. 282
- **getprotounlock()**, p. 283

getprotolock()

Synopsis

```
#include <sys/netLib.h>
int getprotolock( void );
```

Library

netLib.a

Description

getprotolock() locks (acquires exclusive use of) the `_Tcpip_protocols` database.

getprotolock() function should be called before calling **getprotobyname()** and **getprotobynumber()**.

If successful, **getprotolock()** returns 0. Otherwise, **getprotolock()** returns `-1`.

Errors

None.

Example

See **getprotounlock()**, p. 283.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **getprotobyname()**, p. 280
- **getprotobynumber()**, p. 281
- **getprotounlock()**, p. 283

Synopsis

```
#include <sys/netLib.h>
void getprotounlock( void );
```

Library

netLib.a

Description

getprotounlock() unlocks (releases exclusive use of) the **_Tcpi_protocols** database.

getprotounlock() returns nothing.

Errors

None.

Example

The following example prints the number of the UDP protocol and the name of protocol number 6.

```
#include <stdio.h>
#include <types.h>
#include <netdb.h>
int ex_getprotounlock (void)
{
    struct protoent *prot;
    if (getprotolock()) {
        (void)printf("getprotolock() failed\n");
        return(-1);
    }
    prot=getprotobyname("tcp");
    if (prot!=NULL) {
        printf("protocol number of tcp is %d\n", prot->p_proto);
    }
    prot=getprotobyname("udp");
    if (prot!=NULL) {
        printf("protocol number of udp is %d\n", prot->p_proto);
    }
    prot=getprotobynumber(6);
    if (prot!=NULL) {
        printf("protocol name of protocol 6 is %s\n", prot->p_name);
    }
    getprotounlock();
    return(0);
}
```

getprotounlock()

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getprotobyname()**, p. 280
- **getprotobynumber()**, p. 281
- **getprotolock()**, p. 282

Synopsis

```
#include <rpc/netdb.h>

struct rpcent *getrpcbyname( char *name )
```

Library

rpcLib.a

Description

getrpcbyname() retrieves an entry from the **_Rpc_rpc** database using *name* as a search key.

If RPC entry for the specified *name* is found the pointer to the *rpcent* structure containing the entry is returned. If the entry for the specified *name* is not located NULL pointer is returned.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getrpcent()**, p. 287
- **getrpcport()**, p. 288
- **rpc_thread_init()**, p. 676

getrpcbynumber()

Synopsis

```
#include <rpc/netdb.h>
struct rpcent *getrpcbynumber( int number )
```

Library

rpcLib.a

Description

getrpcbynumber() retrieves an entry from the **_Rpc_rpc** database, using *number* as a search key.

If RPC entry for *number* is found, **getrpcbynumber()** returns a pointer to the **rpcent** structure containing the entry. Otherwise, **getrpcbynumber()** returns a NULL pointer.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getrpcbyname()**, p. 285
- **getrpcent()**, p. 287
- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/netdb.h>

struct rpcent *getrpcent( int *position )
```

Library

rpcLib.a

Description

getrpcent() retrieves an entry from the **_Rpc_rpc** database. Entry number specified by the content of the *position* *parameter* is returned.

If more entries are available, the *position* pointer is incremented and the pointer to the specified RPC entry structure is returned. If *position* points to the last entry in the **_Rpc_rpc** database, *position* is not updated and **getrpcent()** returns a NULL.

To retrieve the first entry in the **Rpc_rpc** database, set *position* to 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getrpcent()**, p. 287
- **getrpcport()**, p. 288
- **rpc_thread_init()**, p. 676

getrpcport()

Synopsis

#include <rpc/netdb.h>

int getrpcport(char *host, int prognum, int version, int proto)

Library

rpcLib.a

Description

getrpcport() returns the port number of a remote procedure registered on the computer having the name pointed to by *host*.

prognum specifies the program number of the remote procedure; *versnum* specifies its version number.

proto, which specifies the protocol running on the port, must have a value of **IPPROTO_TCP** or **IPPROTO_UDP**. These constants are defined in the file <netinet.h/in.h>.

If the remote procedure does not exist or is not registered with the portmap daemon on the remote host, **getrpcport()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **pmap_getport()**, p. 492
- **portmap_thread()**, p. 498
- **rpc_thread_init()**, p. 676

Synopsis

```
#include <stdio.h>

char *gets( char *buffer );
```

Library

fsLib.a

Description

gets() reads a line from the standard input and stores it in the buffer pointed to by *buffer*.

The line consists of all characters up to and including the first newline (\n). **gets()** replaces the newline with a NULL (\0) before returning the line.

If successful, **gets()** returns its argument. A NULL pointer indicates an error or end-of-file (EOF). Use **ferror()** or **feof()** to tell which condition occurred.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **strtok()**, p. 808.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.7.7
- **fgets()**, p. 201
- **fputs()**, p. 224
- **fs_init()**, p. 231
- **puts()**, p. 615

getservbyname()

Synopsis

```
#include <netdb.h>

struct servent *getservbyname( const char *name, const char *protocol );
```

Library

netLib.a

Description

getservbyname() retrieves an entry from the **_TcpiP_services** database using a service name as a search key.

getservbyname() can find a service and its aliases, protocol, and port number.

getservbyname() searches the **_TcpiP_services** database sequentially from its start until the function finds one of the following:

- A matching *name* and *protocol*
- A matching *name* when *protocol* is set to NULL
- The end of the database

getservbyname() returns a pointer to a **servent** structure that contains the contents of an entry in the **_TcpiP_services** database. The file **netdb.h** defines the **servent** structure.

If an error occurs or the end of the database is reached, **getservbyname()** returns a NULL pointer.

Note: The return value points to static data that subsequent calls overwrite. Use **getservlock()** and **getservunlock()** to guard the **_TcpiP_services** database and static return value.

Errors

None.

Example

See **inet_aton()**, p. 309.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getservbyport()**, p. 291
- **getservlock()**, p. 293
- **getservunlock()**, p. 294

Synopsis

```
#include <netdb.h>

struct servent *getservbyport( int port, const char *protocol );
```

Library

netLib.a

Description

getservbyport() retrieves an entry from the **_Tcpiip_services** database using a port number as a search key.

getservbyport() can find a service and the service's aliases, protocol, and protocol port number.

getservbyport() searches the services file sequentially from its start until the function finds one of the following:

- A matching *protocol* and *port*.
- A matching *protocol* when *port* is set to 0.
- The end of the **_Tcpiip_services** database.

getservbyport() returns a pointer to a **servent** structure that contains the contents of an entry in the **_Tcpiip_services** database. The file **netdb.h** defines the **servent** structure.

If an error occurs or the end of the database is reached, **getservbyport()** returns a NULL pointer.

Note: The return value points to static data that subsequent calls overwrite. Use **getservlock()** and **getservunlock()** to guard the **_Tcpiip_services** database and static return value.

Errors

None.

Example

The following example prints the name of a service on port 7.

```
struct servent *serv;

:

if (getservlock()) {
    (void)printf("getservlock() failed\n");
    return(-1);
}
if (serv=getservbyport(7, "udp")==NULL) {
    (void)printf("no entry found for port 7 using UDP protocol\n");
```

getservbyport()

```
(void)getservunlock();  
return(-1);  
}  
(void)printf("service on port 7 using UDP is %s\n", serv->s_name);  
(void)getservunlock();  
return(0);
```

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getservbyname()**, p. 290
- **getservlock()**, p. 293
- **getservunlock()**, p. 294

Synopsis

```
#include <sys/netLib.h>
int getservlock( void );
```

Library

netLib.a

Description

getservlock() locks (acquires exclusive use of) the **_Tcpip_services** database.

getservlock() function should be called before calling **getservbyname()** and **getservbyport()**.

If successful, **getservlock()** returns 0. Otherwise, **getservlock()** returns -1.

Errors

None.

Example

See **inet_aton()**, p. 309.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **getservbyname()**, p. 290
- **getservbyport()**, p. 291
- **getservunlock()**, p. 294

getservunlock()

Synopsis

```
#include <sys/netLib.h>
void getservunlock( void );
```

Library

netLib.a

Description

getservunlock() unlocks (releases exclusive use of) the **_Tcpip_services** database.

getservunlock() returns nothing.

Errors

None.

Example

See **inet_aton()**, p. 309.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getservbyname()**, p. 290
- **getservbyport()**, p. 291
- **getservlock()**, p. 293

Synopsis

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

int getsockname( int s, struct sockaddr *name, int *namelen );
```

Library

tcpipLib.a

Description

getsockname() returns the name of the socket *s*.

namelen should initially indicate the amount of space pointed to by *name*. On return, *namelen* contains the actual size of the *name* that was returned. *name* is truncated if the buffer provided is too small.

If successful, **getsockname()** returns 0. Otherwise, **getsockname()** returns -1 and sets *errno*.

Errors

[EBADF]	The descriptor <i>s</i> is not valid.
[ENOTSOCK]	The descriptor <i>s</i> references a file, not a socket.
[ENOBUFS]	Insufficient resources.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.7
- **accept()**, p. 52
- **bprintf()**, p. 84
- **getpeername()**, p. 277
- **socket()**, p. 760

getsockopt()

Synopsis

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

int getsockopt( int s, int level, int optname, char *optval, int *optlen );
```

Library

tcpipLib.a

Description

getsockopt() returns options associated with the socket *s*.

Options, which can exist at multiple protocol levels, are always present at the highest (socket) level.

The option level *level* and the option name must be specified. For options at the socket level, *level* is **SOL_SOCKET**. At any other protocol level, supply the appropriate protocol number for the protocol controlling option.

For example, if an option is to be interpreted by TCP, set *level* to the TCP protocol number. (See **getprotobyname()** on page 280.)

The arguments *optval* and *optlen* identify a buffer in which values of requested options are returned. For **getsockopt()**, *optlen* is a value result parameter initially containing the size of the buffer pointed to by *optval*. On return, *optlen* contains the size of the returned value.

optname and any specified options are passed to the appropriate protocol module for interpretation. The include file **<sys/socket.h>** contains definitions for the following socket level options.

The following options are recognized at the socket level. Except as noted, each may be examined using **getsockopt()** and set using **setsockopt()**

SO_LINGER	Lingers on a close subroutine if data is present. Controls what happens when a thread closes a socket that has an unsent messages queue.
SO_DEBUG	To conserve memory, this option is disabled.
SO_KEEPALIVE	Keeps connections active. Enables or disables periodic transmission of messages on a connected socket. If the connected socket does not respond to the messages, the connection breaks.

SO_DONTROUTE	Does not apply routing on outgoing messages. Indicates that outgoing messages should bypass the standard routing facilities and directs messages to the appropriate network interface, based on the network portion of the destination address.
SO_USELOOPBACK	Uses loopback interface for transmissions.
SO_BROADCAST	Specifies whether transmission of broadcast messages is supported.
SO_REUSEADDR	Specifies that the rules used in validating addresses supplied by a bind subroutine should allow reuse of local addresses.
SO_OOBINLINE	Leaves received out-of-band data (data marked urgent) in line.
SO_SNDBUF	Retrieves send buffer size information.
SO_RCVBUF	Retrieves receive buffer size information.
SO_SNDLOWAT	Retrieves send low-water mark information.
SO_RCVLOWAT	Retrieves receive low-water mark information.
SO_SNDTIMEO	Retrieves send time-out information.
SO_RCVTIMEO	Retrieves receive time-out information.
SO_ERROR	Retrieves information about error status and clear.
SO_TYPE	Retrieves information about a socket type.

If successful, **getsockopt()** returns 0. Otherwise, **getsockopt()** returns -1 and sets *errno*.

Errors

[EBADF]	The descriptor <i>s</i> is not valid.
[ENOTSOCK]	The descriptor <i>s</i> references a file, not a socket.
[ENOPROTOOPT]	The option is unknown at the specified protocol level.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

getsockopt()

References

- IEEE Std 1003.12/D1, §7.5.8
- **getprotobyname()**, p. 280
- **getprotobynumber()**, p. 281
- **setsockopt()**, p. 725
- **socket()**, p. 760

Synopsis

```
#include <time.h>
struct tm *gmtime( const time_t *time );
```

Library

cLib.a

Description

gmtime() converts the time value *time* to a structure.

time is usually obtained using the **time()** function.

gmtime() decomposes *time* and stores it in a structure, **tm**, defined in **<time.h>**. The structure reflects universal coordinated time (UTC), not local time.

The fields of the **tm** structure include:

tm_sec	Seconds(0-59)
tm_min	Minutes(0-59)
tm_hour	Hours(0-23)
tm_mday	Day of month (1-31)
tm_mon	Month (0-11; January = 0)
tm_year	Year (current year since 1900)
tm_wday	Day of week (0-6; Sunday = 0)
tm_yday	Day of year (0-365; January 1 = 0)
tm_isdst	0 if Daylight Saving Time (DST) is not in effect; positive if DST is in effect; negative if no DST information is available.

gmtime() function returns a pointer to the structure result.

gmtime() and **localtime()** use a common, statically allocated buffer for conversion. A call to either function may alter the results of a previous call.

Errors

None.

Example

See **asctime()**, p. 62.

gmtime()

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.12.3.3

Synopsis

```
#include <time.h>
struct tm *gmtime_r( const time_t *time,
                    struct tm *result );
```

Library

cLib.a

Description

gmtime_r() is a reentrant version of **gmtime()**.

Unlike **gmtime()**, the result is not stored in a statically allocated structure. Instead, **gmtime_r()** stores the result in the user-supplied structure pointed to by *result*.

gmtime_r() returns a pointer to the user-supplied structure.

Errors

None.

Example

See **asctime()**, p. 62.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §19.2.3.3
- **gmtime()**, p. 299

heap_list()

Synopsis

```
#include <kadtLib.h>
int heap_list( char **bufp );
```

Library

kadtLib.a

Description

heap_list() provides formatted information about all memory heaps.

The information is part of the **kda_dump()** display.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

heap_list() returns 0.

Errors

None.

Example

The following example shows a sample memory heap listing.

```
OS OPEN>heap_list()
Memory heap    Size          Used          High water mark
0              1810212        937232        937340
----- Total Number of Memory Heaps: 1 -----
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **kda_dump()**, p. 349

Synopsis

```
#include <net/if.h>
int if_attach( struct ifnet *ifp );
```

Library

tcpipLib.a

Description

if_attach() attaches the interface pointed to by *ifp* to the list of active interfaces.

The interface structure *ifp* must contain all necessary information for a particular interface before **if_attach()** is called.

if_attach() returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **socket()**, p. 760

ifconfig()

Synopsis

```
#include <sys/netLib.h>
int ifconfig( char *cmd );
```

Library

netLib.a

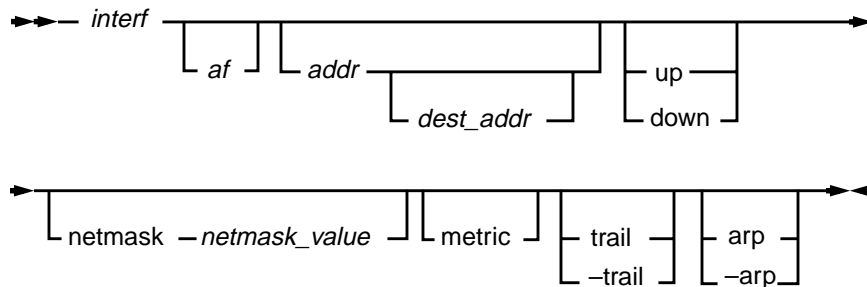
Description

ifconfig() configures or displays the network interface parameters of a network.

ifconfig() can assign an address to a network interface, or configure or display the network interface configuration.

Note: Only one network interface can be associated with a network.

cmd points to a string of the following form:



ifconfig() must be used at OS Open startup to define the network address of each network interface on a machine. Later, **ifconfig()** can redefine the address or other operating parameters of a network interface. OS Open stores a network interface configuration while running; each network interface must be reset at each OS Open restart.

A network interface, specified by *interf*, can receive transmissions using differing protocols, each of which may require separate naming schemes. The calling program must specify *af*, which may affect interpretation of the remaining parameters.

The calling program must specify an interface abbreviation *interf*. The interface abbreviations are:

ent	Standard Ethernet
tok	Token ring
sl	Serial line IP
lo	Loopback

Include a numeral after an abbreviation to specify a particular interface (for example, **tok0**).

ifconfig() currently supports only the **AF_INET** address family. **AF_INET** is defined in `<sys/netLib.h>`. For the **AF_INET** family, *addr* is either a host name present in the host-name database, `_Tcpip_hosts`, or an Internet address expressed in standard Internet dotted decimal notation.

dest_addr specifies the address of the correspondent on the remote end of a point-to-point link.

An **up** command marks an interface as active (up). This parameter is used by default when setting the first address for the interfaces except **sl**. **up** can also enable an interface after a **down** command.

A **down** command marks an interface as inactive (down); OS Open becomes unable to transmit messages using that interface. If possible, **ifconfig()** also resets the interface to disable receipt of messages. Routes that use the interface, however, are not automatically disabled.

netmask specifies how much of the address to reserve for subdividing networks into subnetworks. This parameter can be used only with the **AF_INET** address family. *netmask_value* comprises the network and subnetwork parts of the local address; the subnetwork part is derived from the host field of the local address.

netmask can be specified as a single hexadecimal number beginning with 0x, or in standard Internet dotted decimal notation, or beginning with a name or alias listed in the `_Tcpip_networks` file. *netmask* contains 1s in the bit positions of the 32-bit address that are reserved for the network and subnet work parts, and 0s in the bit positions that specify the host.

metric sets the routing metric number of the interface to the specified value. The default is 0. Higher metric numbers make a route less favorable. Metrics are counted as additional hops to the destination network or host.

-trail and **trail**, respectively, disable and enable trailers on the specified interface.

-arp and **arp**, respectively, disable and enable use of the Address Resolution Protocol to map between network-level addresses and link-level addresses. **arp** is the default.

If successful, **ifconfig()** returns 0. Otherwise, **ifconfig()** returns -1.

Errors

None.

ifconfig()

Example

See **slip_attach()**, p. 756.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- **enet_attach()**, p. 183
- **net_init()**, p. 468
- **ping()**, p. 489
- **slip_attach()**, p. 756

Synopsis

```
#include <ioLib.h>

unsigned char inbyte( unsigned char *port );
```

Library

ioLib.a

Description

inbyte() returns a byte read from the I/O port specified by *port*.

After the byte is read, the PowerPC **eieio** instruction is issued to enforce in-order execution of I/O.

inbyte() returns the byte read from the specified I/O port.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inshort()**, p. 319
- **inword()**, p. 331
- **outbyte()**, p. 481
- **outshort()**, p. 482
- **outword()**, p. 483

inet_addr()

Synopsis

```
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr( const char *b );
```

Library

netLib.a

Description

inet_addr() translates a string, containing an Internet address in dotted decimal notation and pointed to by *b*, into a 32-bit Internet Protocol (IP) address.

If **inet_addr()** fails, it returns 0xffffffff. (Note that 0xffffffff is also the broadcast address).

Errors

None.

Example

The following example retrieves an Internet address.

```
char          *host;
struct sockaddr_in address;
      :
address.sin_addr.s_addr=inet_addr(host);
if (address.sin_addr.s_addr==0xffffffff) {
    (void)printf("invalid address\n");
    return(-1);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inet_aton()**, p. 309
- **inet_lnaof()**, p. 311
- **inet_makeaddr()**, p. 312

Synopsis

```
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton( char *b, struct in_addr *addr );
```

Library

netLib.a

Description

inet_aton() translates a string, containing an Internet address in dotted decimal notation and pointed to by *b*, into a 32-bit Internet Protocol (IP) address.

If successful, **inet_aton()** returns 1 and stores the IP address in the structure pointed to by *addr*. If the address pointed to by *b* is not valid, **inet_aton()** returns 0.

Errors

None.

Example

The following example establishes connection to a host.

```
char          *host;
struct sockaddr_in address;
FILE          *stdout;
              :
              :
(void)printf("enter host name in . notation\n");
(void)fflush(stdout);
(void)gets(host);
(void)bzero((char *)&address, sizeof(address));
if (!inet_aton(host, &(address.sin_addr))) {
    (void)printf("invalid address entered\n");
    return(-1);
}
if (getservlock()) {
    (void)printf("getservlock() failed\n");
    return(-1);
}
address.sin_family=AF_INET;
serv_ent=getservbyname("ftp", "tcp");
if (serv_ent==NULL) {
    (void)printf("getservbyname() failed\n");
    (void)getservunlock();
    return(-1);
}
```

inet_aton()

```
    }
    address.sin_port=serv_ent->s_port;
    (void)getservunlock();
    if ((s=socket(address.sin_family, SOCK_STREAM, 0))==-1) {
        (void)printf("socket() failed\n");
        return(-1);
    }
    if (connect(s, (struct sockaddr *)&address, sizeof(address))) {
        (void)printf("connect() failed\n");
        return(-1);
    }
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inet_addr()**, p. 308
- **inet_lnaof()**, p. 311
- **inet_makeaddr()**, p. 312

Synopsis

```
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_Inaof( struct in_addr inaddr );
```

Library

netLib.a

Description

inet_Inaof() decomposes the Internet Protocol (IP) address *inaddr* and returns its local portion.

inet_Inaof() handles class A, B, and C IP address formats.

Errors

None.

Example

The following example prints the local portion of an IP address.

```
unsigned long val;
struct in_addr address;

:

val=inet_Inaof(s);
(void)inet_ntoa_r(val, address);
(void)printf("local portion is %s\n", address);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inet_addr()**, p. 308
- **inet_makeaddr()**, p. 312
- **inet_netof()**, p. 313

inet_makeaddr()

Synopsis

```
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr inet_makeaddr( unsigned long net,
                             unsigned long local_addr );
```

Library

netLib.a

Description

inet_makeaddr() combines an Internet Protocol (IP) network number and a local network address and returns a new IP address.

net contains an IP network number; *local_addr* contains a local network address.

Errors

None.

Example

See **inet_network()**, p. 314.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inet_addr()**, p. 308
- **inet_lnaof()**, p. 311
- **inet_netof()**, p. 313
- **inet_network()**, p. 314

Synopsis

```
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_netof( struct in_addr inaddr );
```

Library

netLib.a

Description

inet_netof() decomposes the Internet Protocol (IP) address *inaddr* and returns its network portion.

inet_netof() handles class A, B, and C IP address formats.

Errors

None.

Example

The following example prints the network portion of an IP address.

```
unsigned long val;
struct in_addr address;

:

val=inet_netof(s);
(void)inet_ntoa_r(val, address);
(void)printf("network portion is %s\n", address);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inet_addr()**, p. 308
- **inet_lnaof()**, p. 311
- **inet_makeaddr()**, p. 312
- **inet_network()**, p. 314

inet_network()

Synopsis

```
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_network( const char *cp );
```

Library

netLib.a

Description

inet_network() translates a string, containing an Internet address in dotted decimal notation and pointed to by *cp*, into an Internet network number.

If unsuccessful, **inet_network()** returns **INADDR_NONE**, which is defined in **<netinet/in.h>**.

Errors

None.

Example

The following example retrieves a network address and name.

```
#include <stdio.h>
#include <types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>

int netaddress(char *s, struct sockaddr_in *sin, char *name)
{
    unsigned long val;
    int rc=0;
    struct netent *np;

    val=inet_network(s);
    if (val!=INADDR_NONE) {
        sin->sin_family=AF_INET;
        sin->sin_addr=inet_makeaddr(val, INADDR_ANY);
        (void)strcpy(name, s);
    }
    if (getnetlock()) {
        (void)printf("getnetlock() function failed\n");
        return(-1);
    }
    np=getnetbyname(s);
    if (np) {
```

```
sin->sin_family=np->n_addrtype;
sin->sin_addr=inet_makeaddr(np->n_net, INADDR_ANY);
(void)strcpy(name, np->n_name);
} else {
    rc=-1;
}
(void)getnetunlock();
return(rc);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inet_addr()**, p. 308
- **inet_aton()**, p. 309
- **inet_ntoa()**, p. 316
- **inet_ntoa_r()**, p. 317

inet_ntoa()

Synopsis

```
#include <netinet/in.h>
#include <arpa/inet.h>
char *inet_ntoa( struct in_addr in );
```

Library

netLib.a

Description

inet_ntoa() translates the Internet Protocol (IP) address *in* into an ASCII string representing the Internet address in dotted decimal notation.

Errors

None.

Example

The following example prints an Internet address in dotted decimal notation.

```
struct in_addr in;
    :
    :
in.s_addr=0xFFFF4321;
(void)printf("packet received from %s\n", inet_ntoa(in));
```

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inet_addr()**, p. 308
- **inet_aton()**, p. 309
- **inet_network()**, p. 314
- **inet_ntoa_r()**, p. 317

Synopsis

```
#include <netinet/in.h>
#include <arpa/inet.h>

void inet_ntoa_r( struct in_addr in, char *b );
```

Library

netLib.a

Description

inet_ntoa_r() is a reentrant version of **inet_ntoa()**.

inet_ntoa_r() translates the Internet Protocol (IP) address *in* to an ASCII string representing the Internet address in dotted decimal format.

inet_ntoa_r() stores the resulting ASCII string in the structure pointed to by *b*, which should contain at least 16 characters.

inet_ntoa_r returns nothing.

Errors

None.

Example

The following example prints an Internet address in dotted decimal format.

```
struct in_addr in;
char          *local_addr;
              :
              :
local_addr=(char*)malloc(40);
in.s_addr=0xffff4321;
(void)inet_ntoa_r(in, local_addr);
(void)printf("packet received from %s\n", local_addr);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inet_addr()**, p. 308
- **inet_aton()**, p. 309
- **inet_network()**, p. 314
- **inet_ntoa()**, p. 316

info_rtable()

Synopsis

```
#include <net/route.h>
```

```
int info_rtable( int op, char *where, int *given, int arg, int *needed );
```

Library

tcplib.a

Description

info_rtable() retrieves routing information from the TCP/IP library routing table.

The *op* argument, which specifies the operation to be performed, can be set to either **INFO_RT_FLAGS** or **INFO_RT_DUMP**; these constants are defined in **<net/route.h>**. If *op* parameter is set to **INFO_RT_FLAGS**, only entries matching flag specified by *arg* parameter are copied.

where points to the address of the buffer where the routing information is copied. If *where* is set to NULL, no information is copied and *needed* returns the amount of space required to copy the routing information. *given* specifies the size of the buffer pointed to by *where*.

If successful, **info_rtable()** returns 0. Otherwise, **info_rtable()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>op</i> does not equal INFO_RT_FLAGS or INFO_RT_DUMP .
----------	--

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **ioctl()**, p. 332
- **socket()**, p. 760

Synopsis

```
#include <ioLib.h>
unsigned short inshort( unsigned short *port );
```

Library

ioLib.a

Description

inshort() returns a halfword read from the I/O port specified by *port*.

After the halfword is read, the PowerPC **ei** instruction is issued to enforce in-order execution of I/O.

inshort() returns the halfword read from the specified I/O port.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inbyte()**, p. 307
- **inword()**, p. 331
- **outbyte()**, p. 481
- **outshort()**, p. 482
- **outword()**, p. 483

int_disable()

Synopsis

```
#include <flih.h>
void int_disable( event_t event );
```

Library

rtxLib.a

Description

int_disable() disables the event named by *event*.

Refer to the file **<flih.h>** for values that can be specified using **int_disable()**.

int_disable() returns nothing.

Note: **int_disable()** is not available in PowerPC 6xx processors.

Note: In the POWER Architecture*, **int_disable()** clears the bit in the EIM0 register corresponding to the specified event.

Errors

None.

Example

See **int_query()**, p. 329.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **int_enable()**, p. 321

Synopsis

```
#include <flih.h>

void int_enable( event_t event );
```

Library

rtxLib.a

Description

int_enable() enables the event named by *event*.

Refer to the file **<flih.h>** for values that can be specified using **int_enable()**.

int_enable() function returns nothing.

Note: **int_enable()** is not available in PowerPC 6xx processors.

Note: In the POWER Architecture, **int_enable()** sets the bit in the EIM0 register corresponding to the specified event.

Errors

None.

Example

See **int_install()**, p. 327.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **int_disable()**, p. 320

int_flihdisable()

Synopsis

```
#include <ioLib.h>

void int_flihdisable( event_t event );
```

Library

ioLib.a

Description

int_flihdisable() calls the user-supplied disable function previously registered using **int_flihregister()**. **int_flihdisable()** is called in the beginning of all OS Open external first level interrupt handlers (FLIHs).

int_flihdisable() can be called from user-supplied FLIHs to support nesting of external interrupts. User-specified functions can disable events and enable external interrupts, and prioritize the interrupts.

In FLIHs, this function should be paired with **int_flihenable()**.

Errors

None.

Example

At the start of the following sample FLIH **async_flih()**, **int_flihdisable()** is called; before returning from **async_flih()**, **int_flihenable()** is called. **async_flih()** handles interrupts from two sources (serial ports), so that **int_flihdisable()** and **int_flih_enable()** are called with different parameters.

```
static void async_flih(void *arg)
{
    char          uart_reg;
    char          buffer[ASYNC_FIFO_SIZE];
    unsigned long  rc, temp=false;
    asyncdds_t     *dds;

    if (arg==(void *)SAND_COM1_IRQ) {
        (void)int_flihdisable(SAND_COM1_IRQ);
        dds=async_port1;
    } else {
        (void)int_flihdisable(SAND_COM2_IRQ);
        dds=async_port2;
    }
    /* device driver interrupt handling code */

    if (arg==(void *)SAND_COM1_IRQ) {
```

```
        (void)int_flihenable(SAND_COM1_IRQ);
    } else {
        (void)int_flihenable(SAND_COM2_IRQ);
    }
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- [int_flihenable\(\)](#), p. 324
- [int_flihregister\(\)](#), p. 325

int_flihenable()

Synopsis

```
#include <ioLib.h>

void int_flihenable( event_t event );
```

Library

ioLib.a

Description

int_flihenable() calls the user-supplied enable function previously registered using **int_flihregister()**. **int_flihenable()** is called at the end of all OS Open external first level interrupt handlers (FLIHs).

int_flihenable() can also be called from user-supplied FLIHs to support nesting of external interrupts. User-specified functions can disable events and enable external interrupts, and prioritize the interrupts.

In FLIHs, this function should be paired with **int_flihdisable()**.

Errors

None.

Example

See **int_flihdisable()**, p. 322.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **int_flihdisable()**, p. 322
- **int_flihregister()**, p. 325

Synopsis

```
#include <ioLib.h>

void int_flihregister(void(*enablefunc)(event_t event), void
(*disablefunc)(event_t event));
```

Library

ioLib.a

Description

int_flihregister() registers the user-supplied functions *enablefunc()* and *disablefunc()* to be called when **int_flihenable()** and **int_flihdisable()** are called.

If *enablefunc()* or *disablefunc()* is set to **NULL**, user-supplied functions are unregistered.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **int_flihenable()**, p. 324
- **int_flihdisable()**, p. 322

int_getid()

Synopsis

```
#include <flih.h>
pthread_t int_getid( void );
```

Library

rtxLib.a

Description

int_getid() returns the thread ID of the currently executing thread; if called from an external interrupt handler context, **int_getid()** returns 0.

int_getid() can determine whether an external interrupt handler is running.

When there are no ready threads, OS Open executes an idle loop inside the kernel. If an exception occurs while the idle loop executes and an exception handler calls **int_getid()**, -1 is returned to indicate that no thread or external interrupt was running when the exception occurred.

If issued by an exception handler, **int_getid()** returns the thread ID of the thread that caused the exception. If the exception occurred in an external or time-based interrupt handler, **int_getid()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **int_install()**, p. 327
- **int_query()**, p. 329

Synopsis

```
#include <flih.h>

int int_install( event_t event, flih_t *new_flih, flih_t *old_flih );
```

Library

rtxLib.a

Description

int_install() installs a first-level interrupt handler (FLIH) for the event named by *event*. FLIHs can be installed for all events listed in the file **<flih.h>**.

When installed, the EVENT_EXT_ALL_NP event is called for all external interrupts irrespective of other event handlers installed to handle external interrupts.

When EVENT_EXT_ALL_NP is removed, external interrupt events are handled by their appropriate handlers.

EVENT_EXT_ALL_NP allows user application control over external interrupt prioritization on processors such as PowerPC 400Series devices.

Note: For PowerPC 6xx processors, EVENT_EXT_ALL_NP is identical to EVENT_EXT0_NP, since those processors have only one external interrupt.

If *new_flih* is NULL, the old interrupt handler is removed for that specified event. If *new_flih* is non-NULL, it points to a structure **flih_t** containing the following fields:

flih_function	A pointer to a function to be invoked when the event occurs.
flih_stack	A pointer to the top of the stack, obtained by adding the size of the stack to the address obtained when memory was allocated for the stack. Note that the stack grows toward lower addresses.
arg	A (void *) user-defined value passed to flih_function .

If an event defined in **<flih.h>** occurs and no interrupt handler is installed for the event, the OS Open operating system defaults to the user-supplied panic routine.

If *old_flih* is not NULL, the previous values of *flih_function*, *flih_stack*, and *arg* are stored in the structure pointed to by *old_flih*.

If successful, **int_install()** returns 0. Otherwise, **int_install()** returns -1 and sets *errno*.

Errors

[EINVAL]	The value of <i>event</i> is not valid.
----------	---

int_install()

Example

The following example installs a FLIH for an event and enables the event.

```
#include <flih.h>
#include <stddef.h>
#include <memLib.h>

void _sample_flih(void);
#define SAMPLE_STACK 1024
#define SAMPLE_INTR_LEVEL EVENT_EXT28_NP

int sample_install(void)
{
    flih_t sam_flih;
    int rv;
    void *t;

    /* Initialize flih structure */
    t = memheap_alloc(APPL_HEAP, SAMPLE_STACK);
    if(t == NULL)
        return(-1);
    sam_flih.flih_stack = (char *)t + SAMPLE_STACK;
    sam_flih.flih_function = (void *)_sample_flih;
    sam_flih.arg = (void *)SAMPLE_INTR_LEVEL;
    /* Install the flih */
    rv = int_install(SAMPLE_INTR_LEVEL, &sam_flih, NULL);
    if(rv < 0)
    {
        memheap_free(t);
        return(-1);
    }
    /* Enable the event */
    int_enable(SAMPLE_INTR_LEVEL);

    return(0);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- `int_query()`, p. 329

Synopsis

```
#include <flih.h>

int int_query( event_t event,
              flih_t *flih );
```

Library

rtxLib.a

Description

int_query() returns information about the first level interrupt handler (FLIH), if any, for the event named by *event*.

Refer to the file **<flih.h>** for values that can be specified for *event*.

flih points to a **flih_t** structure that contains the following values:

flih_function	A pointer to a function that is invoked when the event occurs.
flih_stack	A pointer to the first stack location, which is obtained by allocating memory and adding the size of the stack.
arg	A (void *) value of user-defined significance passed to <i>flih_function</i> .

If no FLIH is installed for *event*, each field in the **flih_t** structure is assigned NULL.

If successful, **int_query()** returns 0. Otherwise, **int_query()** returns -1 and sets *errno*.

Errors

[EINVAL]	The value of <i>event</i> is not valid, or a NULL pointer was specified for <i>flih</i>
----------	---

Example

The following example is a function that removes a FLIH for an event and disables the event.

```
#include <flih.h>
#include <stddef.h>

int remove_flih(event_t event)
{
    flih_t flih_info;
    int rv;

    /* Check to see if a flih has been installed for this event */
    rv = int_query(event, &flih_info);
```

int_query()

```
    if(rv < 0)
        return(-1);
    /* Disable the event */
    int_disable(event);
    if(flih_info.flih_function == NULL) /* no flih installed */
        return(0);
    /* remove the flih */
    rv = int_install(event, NULL, &flih_info);
    if(rv < 0)
        return(-1);

    return(0);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- `int_install()`, p. 327

Synopsis

```
#include <ioLib.h>
unsigned long inword( unsigned long *port );
```

Library

ioLib.a

Description

inword() reads a word from the I/O port specified by *port*.

After the word is read, the PowerPC **eieio** instruction is issued to enforce in-order execution of I/O.

inword() returns the word read from the specified I/O port.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inbyte()**, p. 307
- **inshort()**, p. 319
- **outbyte()**, p. 481
- **outshort()**, p. 482
- **outword()**, p. 483

ioctl()

Synopsis

```
#include <unistd.h>
#include <sys/ioctl.h>

int ioctl( int filedes, int cmd, ... );
```

Library

devLib.a

Description

ioctl() uses the open file descriptor *filedes* to issue device-specific control commands.

In general, values of *cmd* are device-specific, and individual device driver documentation and **<sys/ioctl.h>** must be consulted.

If successful, **ioctl()** returns 0. Otherwise, **ioctl()** returns `-1` and sets *errno*.

Note: A data value of 1 sets conditions and a data value of 0 clears conditions for **FIONBIO** and **FIOASYNC**.

Errors

[EBADF]	<i>filedes</i> is not a valid file descriptor.
[EINVAL]	<i>cmd</i> is undefined for the device associated with <i>filedes</i> .

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **open()**, p. 478

Synopsis

```
#include <ctype.h>
int isalnum( int c );
```

Library

cLib.a

Description

isalnum() returns nonzero (true) for any character *c* for which **isalpha()** or **isdigit()** return true.

Otherwise, **isalnum()** returns 0 (false).

isalpha() returns true for uppercase and lowercase letters. **isdigit()** returns true for decimal digits.

c must be representable as an **unsigned char** or EOF.

Errors

None.

Example

The following example counts the types of characters in a file.

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

void count_char(FILE *file)
{
    long count[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    char c;

    /* Reset the file position to the beginning */
    rewind(file);
    /* Read one byte at a time until the end of file is reached */
    while(!feof(file))
    {
        c = fgetc(file);
        if(c == EOF)
            break;
        if(isalnum(c))
            count[0]++;
        if(isalpha(c))
            count[1]++;
        if(iscntrl(c))
            count[2]++;
    }
}
```

isalnum()

```
        if(isdigit(c))
            count[3]++;
        if(isgraph(c))
            count[4]++;
        if(islower(c))
            count[5]++;
        if(isprint(c))
            count[6]++;
        if(ispunct(c))
            count[7]++;
        if(isspace(c))
            count[8]++;
        if(isupper(c))
            count[9]++;
    }
    /* print the results */
    printf("Alphanumeric characters: %d\n", count[0]);
    printf("Alphabetic characters: %d\n", count[1]);
    printf("Control characters: %d\n", count[2]);
    printf("Decimal digit characters: %d\n", count[3]);
    printf("Printable, except space characters: %d\n",
        count[4]);
    printf("Lower case characters: %d\n", count[5]);
    printf("Printable characters: %d\n", count[6]);
    printf("Punctuation characters: %d\n", count[7]);
    printf("White space characters: %d\n", count[8]);
    printf("Upper case characters: %d\n", count[9]);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.1.1

Synopsis

```
#include <ctype.h>
int isalpha( int c );
```

Library

cLib.a

Description

isalpha() returns nonzero (true) for any character *c* for which **islower()** or **isupper()** return true.

Otherwise, **isalpha()** returns 0 (false).

islower() returns true for lowercase letters; **isupper()** returns true for uppercase letters.

c must be representable as an **unsigned char** or EOF.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.1.2

isatty()

Synopsis

```
#include <ttyLib.h>
int isatty(int filedes);
```

Library

ttyLib.a

Description

isatty() returns 1 if *filedes* is a valid file descriptor associated with a terminal. Otherwise, **isatty()** returns 0 (false).

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §4.7.2

Synopsis

```
#include <ctype.h>
int iscntrl( int c );
```

Library

cLib.a

Description

iscntrl() returns nonzero (true) for any character *c* that is a control character.

Otherwise, **iscntrl()** returns 0 (false).

c must be representable as an **unsigned char** or EOF.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.1.3

isdigit()

Synopsis

```
#include <ctype.h>
int isdigit( int c);
```

Library

cLib.a

Description

isdigit() returns nonzero (true) for any character *c* that is a decimal digit.

Otherwise, **isdigit()** returns 0 (false).

c must be representable as an **unsigned char** or EOF.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.1.4

Synopsis

```
#include <ctype.h>
int isgraph( int c );
```

Library

cLib.a

Description

isgraph() returns nonzero (true) for any character *c* that is printable except the space character (' ').

Otherwise, **isgraph()** returns 0 (false).

c must be representable as an **unsigned char** or EOF.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.1.5

islower()

Synopsis

```
#include <ctype.h>
int islower( int c );
```

Library

cLib.a

Description

islower() returns nonzero (true) for any character *c* that is a lowercase letter.

Otherwise, **islower()** returns 0 (false).

c must be representable as an **unsigned char** or EOF.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.1.6

Synopsis

```
#include <ctype.h>
int isprint( int c);
```

Library

cLib.a

Description

isprint() returns nonzero (true) for any character *c* that is a printable character, including the space character (' ').

Otherwise, **isprint()** returns 0 (false).

c must be representable as an **unsigned char** or EOF.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.1.7

ispthreadid()

Synopsis

```
#include <kadtLib.h>
int ispthreadid( pthread_t thread );
```

Library

kadtLib.a

Description

ispthreadid() returns nonzero (true) if *thread* is a valid thread ID.
Otherwise, **ispthreadid()** returns 0 (false).

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <ctype.h>
int ispunct( int c );
```

Library

cLib.a

Description

ispunct() returns nonzero (true) for any character *c* for which **isalnum()** returns 0 (false) or that is not a space character (' ').

Otherwise, **ispunct()** returns 0 (false).

c must be representable as an **unsigned char** or EOF.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.1.8

isspace()

Synopsis

```
#include <ctype.h>
int isspace( int c );
```

Library

cLib.a

Description

isspace() returns nonzero (true) for any character *c* that is one of the standard whitespace characters.

The whitespace characters include space (' '), form feed (\f), newline (\n), carriage return (\r), horizontal tab (\t), and vertical tab (\v).

Otherwise, **isspace()** returns 0 (false).

c must be representable as an **unsigned char** or EOF.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.1.9

Synopsis

```
#include <kadtLib.h>
int issuspended( pthread_t thread )
```

Library

kadtLib.a

Description

issuspended() returns nonzero (true) if *thread* is suspended at a breakpoint or an error.

If *thread* is not suspended or is not a valid thread ID, **issuspended()** returns 0 (false).

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

istgid()

Synopsis

```
#include <kadtLib.h>
int istgid(tg_t tgid);
```

Library

kadtLib.a

Description

istgid() returns nonzero (true) if *tgid* is a valid thread group ID. Otherwise, **istgid()** returns 0 (false).

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **pthread_tgcreate_np()**, p. 598
- **pthread_tgdestroy_np()**, p. 599
- **pthread_whattg_np()**, p. 608

Synopsis

```
#include <ctype.h>
int isupper( int c );
```

Library

cLib.a

Description

isupper() returns nonzero (true) for any character *c* that is an uppercase letter.

Otherwise, **isupper()** returns 0 (false).

c must be representable as an **unsigned char** or EOF.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.1.10

isxdigit()

Synopsis

```
#include <ctype.h>
int isxdigit( int c );
```

Library

cLib.a

Description

isxdigit() returns nonzero (true) for any character *c* that is a hexadecimal digit.

Otherwise, **isxdigit()** returns 0 (false).

c must be representable as an **unsigned char** or EOF.

Errors

None.

Example

See **flockfile()**, p. 205.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.1.11

Synopsis

```
#include <kadtLib.h>
int kda_dump( char *buf );
```

Library

kadtLib.a

Description

kda_dump() formats and displays the kernel data area (KDA).

kda_dump() displays threads, mutexes, condition variables, semaphores, message queues, timers and signals. If the value of the pointer *buf* is NULL, the output is sent to the standard output. Otherwise, the information is stored in the buffer pointed to by *buf*.

kda_dump() returns -1 if the KDA is not found. Otherwise, **kda_dump()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **cond_list()**, p. 130
- **flih_list()**, p. 203
- **heap_list()**, p. 302
- **library_list()**, p. 382
- **mq_list()**, p. 438
- **mutex_list()**, p. 463
- **pool_list()**, p. 497
- **semaphore_list()**, p. 711
- **signal_list()**, p. 747
- **thread_info_list()**, p. 883
- **timer_list()**, p. 895

kill()

Synopsis

```
#include <sys/types>
#include <signal.h>
int kill( pid_t pid, int sig );
```

Library

rtxLib.a

Description

kill() sends a signal *sig* to a process.

Because OS Open consists of only one process, **kill()** directs *sig* to the system environment. Signals are delivered to a thread in the following situations:

- The thread has called **sigwait()** for the pending signal.
- The thread is unblocked for the signal and is entering a system call that would block, such as **sem_lock()** or **pthread_cond_wait()**.

Note: Unlike **pthread_kill()**, **kill()** does not interrupt a thread already waiting on a blocking system call, and *sig* is not consumed by the interrupted signal call. *sig* remains pending until consumed by **sigwait()**.

If multiple threads are eligible to receive *sig*, it is delivered to only one thread. Which eligible thread is selected is unspecified.

If *sig* equals 0, error checking is performed but *sig* is not sent.

For OS Open, valid values for *pid* are 0 or, for POSIX compatibility, a return value or the negation of a return value from **getpid()**.

If successful, **kill()** returns 0. Otherwise, **kill()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>sig</i> is an unsupported signal number or is not valid.
[ESRCH]	No process or process group matches <i>pid</i> .

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, §3.3.2

Synopsis

```
#include <stdlib.h>
long int labs( long int j );
```

Library

cLib.a

Description

labs() returns the absolute value of the long integer operand *j*.

Note: A **long int** can hold a negative number whose absolute value is too large to express as a **long int**. The minimum value allowed for *j* is **LONG_MIN** + 1.

Errors

None.

Example

See **atol()**, p. 74.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.6.3
- **abs()**, p. 51

ld()

Synopsis

```
#include <ldrLib.h>
int ld( char *pathname );
```

Library

ldrLib.a

Description

ld() loads an executable program into memory, relocates it, links it with other loaded programs, and updates the resident symbol table with the program's external symbols.

The symbol library needs to be initialized prior to loading any files. The library can be initialized by calling **statsym_init()**. If garbage collection is turned on, then some functions (or function descriptors in the case of the XL C compiler family) required by loaded modules are removed from the initial load image. To prevent this, an export list has to be identified during the link process.

pathname must name a readable file in executable format. A loaded object specifier, *ldname*, created by stripping *pathname* of directories, can be used on subsequent loader calls such as **ldrFree()**, **ldrUnload()**, and **ldrEntry_get()**.

If an object of the same *ldname* was previously loaded, **ld()** unloads it before loading the file named by *ldname*.

Error and warning messages returned by **ld()** are written to the standard output.

If successful, **ld()** returns 0. If there are warning messages, **ld()** returns 1. If one or more errors occur, **ld()** returns -1 and the loaded program is not executable.

Errors

None.

Example

The following example loads the file **test1.ld** from the directory **/fat/test/ld**, gets the program entry point for a function pointer, and calls the loaded function. Note that the media device driver and the file allocation table (FAT) file system driver are assumed to be available.

```
#include <ldrLib.h>

int run_test1(void) {
    int rc;
    int (*func_p)();

    /* Load test1 program. */
    rc = ld("/fat/test/ld/test1.ld");
```



```
if (rc == -1 && !ldrIsexec("test1.ld"))
    return -1;
/* Free memory used for loader table */
ldrFree("test1.ld");
/* Get the program entry point. */
func_p = (int (*)(void))ldrEntry_get("test1.ld");
/* Call function test1 through pointer */
return (*func_p)();
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **ldrEntry_get()**, p. 359
- **ldrFree()**, p. 377
- **ldrIsexec()**, p. 360
- **ldrUnload()**, p. 381

ldexp()

Synopsis

```
#include <math.h>
double ldexp( double x, int exp );
```

Library

mathLib.a

Description

ldexp() multiplies the floating-point number *x* by 2 raised to the *exp* power.

If the result is out of range, a range error occurs and **ldexp()** sets *errno*.

Errors

[ERANGE]	The result is out of range.
----------	-----------------------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.4.3

Synopsis

```
#include <stdlib.h>

ldiv_t ldiv( long int numer,
             long int denom );
```

Library

cLib.a

Description

ldiv() computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*.

If the result can be represented, **ldiv()** returns a structure containing two long integer values: the quotient *quot* and the remainder *rem*, such that:

$$quot * denom + rem = numer$$

Otherwise, **ldiv()** returns NULL.

Errors

None.

Example

See **atol()**, p. 74.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.6.4
- **div()**, p. 176

ldrDup_name()

Synopsis

```
#include <ldrLib.h>

int ldrDup_name( char *ldname, char *name, size_t len, void **prev );
```

Library

ldrLib.a

Description

ldrDup_name() returns the names of duplicate-named symbols from the loaded object identified by *ldname*.

The name of the duplicate symbol is copied into the character buffer pointed to by *name*. The *len* parameter specifies the maximum number of characters to copy. If a symbol name is longer than *len*, it is truncated without an indication.

prev points to a pointer to the most recent duplicate symbol. *prev* should point to a NULL pointer the first time **ldrDup_name()** is called. Subsequent calls to **ldrDup_name()** update *prev* so that the next duplicate symbol is returned. After the last duplicate symbol is returned, *prev* is set to (void *)-1.

When there are no more duplicate symbols in the loaded object, *name* is set to a NULL string ('\0').

If successful, **ldrDup_name()** returns 0. Otherwise, **ldrDup_name()** returns -1.

Errors

None.

Example

The following function loads patches in a file whose name is passed as an input parameter. It is assumed that all “patchable” functions are called through pointers, and that the pointers are named by concatenating “_p” to the function names. The functions and the corresponding pointers that are to be replaced are found by looking at duplicate symbol names.

```
#include <ldrLib.h>
#include <symLib.h>
#include <sys/types.h>
#include <fcntl.h>

int load_patches(char *patch_file) {
    int fd;
    ldrInfo_t li;          /* Loader information structure */
    char symn[256];
    void *prev = NULL;
    unsigned long newfunc, *funcp;
```

```
/* Initialize symbol table in case this is the first time */
statsym_init(STATSYM_INIT_NO_HID_EXT | STATSYM_INIT_NO_BY_ADDR);
/* Load patch file */
fd = open(patch_file,O_RDONLY);    /* Open patch file for reading */
if (fd < 0) {
    perror("Error opening patch file");
    return -1;
}
if (ldrQuery(fd,&li) != 0) {        /* Get loader information */
    ldrMsg(NULL,NULL,0);           /* Print error loader error msgs */
    close(fd);
    return -1;
}
if (ldrLoad(fd,patch_file,&li) < 0) { /* Load file */
    ldrMsg(NULL,NULL,0);           /* Print error loader error msgs */
    close(fd);
    return -1;
}
statsym_update(&li);               /* Update symbol table */
/* Update patched functions */
while (prev != (void *)-1) {
    /* Find duplicate named symbols */
    if (ldrDup_name(patch_file, symn, sizeof(symn), &prev) != 0) {
        close(fd);
        return -1;
    }
    if ( strlen(symn) ) {           /* If valid name returned */
        newfunc = statsym_find_ldsym(symn, &li);
        strcat(symn,"_p");         /* global function pointer name */
        funcp = (unsigned long *)statsym_find_ext(symn);
        /* If both function pointer and new function are found */
        if (funcp && newfunc)
            *funcp = newfunc;      /* Patch function pointer */
    }
}
/* Clean up temporary storage */
close(fd);
ldrFree(patch_file);              /* Clean up after loader */
statsym_remove(NULL);             /* Clean up symbol table */
return 0;
}
```

IdrDup_name()

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **IdrLink()**, p. 361
- **IdrLoad()**, p. 363

Synopsis

```
#include <ldrLib.h>

unsigned long ldrEntry_get( char *ldname );
```

Library

ldrLib.a

Description

ldrEntry_get() returns the entry address of the loaded object named by *ldname*.

The returned address is the address of the function descriptor (for XCOFF) or address of the entry function (ELF) specified using the **-e** flag when the file containing the loaded object was created.

If no entry was specified when the file was linked, the entry is 0xffffffff.

If **ldrEntry_get()** fails, **ldrEntry_get()** returns 0.

Errors

None.

Example

See **ld()**, p. 352.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **ld()**, p. 352
- **ldrLink()**, p. 361
- **ldrLoad()**, p. 363

Idrlsexec()

Synopsis

```
#include <ldrLib.h>
int Idrlsexec( char *ldname );
```

Library

ldrLib.a

Description

Idrlsexec() returns the executable status of the loaded object named by *ldname*.

If the loaded object is executable, **Idrlsexec()** returns 1 (true). Otherwise, **Idrlsexec()** returns 0 (false).

Errors

None.

Example

See **ld()**, p. 352.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **ld()**, p. 352
- **ldrLink()**, p. 361
- **ldrLoad()**, p. 363

Synopsis

```
#include <ldrLib.h>

int ldrLink( void *objp, char *ldname, ldrInfo_t *info );
```

Library

ldrLib.a

Description

ldrLink() relocates the loaded object image pointed by *objp* to the loaded location. The object image must be in XCOFF format for XCOFF-based OS Open or ELF format for ELF-based OS Open.

objp points to an XCOFF or ELF executable object file which has been loaded into memory. *ldname* points to an object name that can be used on subsequent loader calls, such as **ldrMsg()** and **ldrUnlink()**. *info* points to a structure of type **ldrInfo_t** that must be initialized by calling **ldrQimage()** for the object image. After calling **ldrQimage()** the *info* structure may be modified by calling **ldrMoveData()** and **ldrMoveSym()**. If *info* is a NULL pointer, it is ignored.

If successful, **ldrLink()** returns 0. If duplicate symbols are present, **ldrLink()** returns 1 and saves a warning message for later retrieval using **ldrMsg()** or **ldrDup_name()**. Otherwise, **ldrLink()** returns -1 and saves error message for retrieval by **ldrMsg()**.

Errors

None.

Example

The following example links a loaded read-only object file image.

```
#include <stdio.h>
#include <ldrLib.h>
#include <symLib.h>

/*
 * linkit - link a loaded object file image
 */
int linkit(void *objp, char *ldname) {
    int rc;
    ldrInfo_t li;

    rc = ldrQimage(objp,&li);    /* Query the storage requirements */
    if (rc != 0) {
        ldrMsg(NULL,NULL,0);    /* Print any errors that may have occurred */
        return -1;
    }
}
```

IdrLink()

```
rc = IdrMoveData(objp,NULL,&li); /* Move data to rw memory */
if (rc != 0) {
    IdrMsg(NULL,NULL,0);          /* Print error message */
    return -1;
}
rc = IdrMoveSym(objp,NULL,&li); /* Move symbols to rw memory */
if (rc != 0) {
    IdrMsg(NULL,NULL,0);          /* Print error message */
    return -1;
}
rc = IdrLink(objp,ldname,&li);    /* Link object file image */
IdrMsg(ldname,NULL,0);           /* Display any messages */
if (rc >= 0) {
    statsym_update(&li);          /* Update symbol table */
    IdrResolve();                 /* Resolve references to the newly linked image */
}
return rc;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **IdrMsg()**, p. 368
- **IdrQimage()**, p. 369
- **IdrLoad()**, p. 363

Synopsis

```
#include <ldrLib.h>

int ldrLoad( int fd, char *ldname, ldrInfo_t *info );
```

Library

ldrLib.a

Description

ldrLoad() loads the open file *fd* into memory and relocates it to the loaded location. *fd* must be in XCOFF format for XCOFF-based OS Open or ELF format for ELF-based OS Open.

ldname points to an object name that can be used on subsequent loader calls, such as **ldrMsg()** and **ldrUnload()**. *info* points to a structure of type **ldrInfo_t** that must be initialized with the section sizes. The structure should be initialized by calling **ldrQuery()** for the file.

ldrLoad() allocates storage for any sections in the **ldrInfo_t** structure that have *addr* set to 0. If any *addr* fields are not 0, the sections are loaded at the specified addresses.

If successful, **ldrLoad()** sets the *addr* fields in the **ldrInfo_t** structure to the addresses where the sections were loaded.

The *size* fields in **ldrInfo_t** must be no smaller than the value returned by calling **ldrQuery()** on the file. However, if the *symbol.size* field is set to zero, **ldrLoad()** does not load the symbol table from the object file.

If successful, **ldrLoad()** returns 0. If duplicate symbols are present, **ldrLoad()** returns 1 and saves a warning message for later retrieval using **ldrMsg()** or **ldrDup_name()**. Otherwise, **ldrLoad()** returns -1 and saves error message for retrieval by **ldrMsg()**.

Errors

None.

Example

The following example loads a file into memory.

```
#include <stdio.h>
#include <string.h>
#include <types.h>
#include <fcntl.h>
#include <ldrLib.h>
#include <symLib.h>

/*
```

IdrLoad()

```
* Id - load a file into memory
*/
int Idr(char *pathname) {
    int rc=0, fd;
    char fname[128];
    char error[256];
    char *ldname;
    ldrInfo_t li;

    fd = open(pathname,O_RDONLY);
    if (fd < 0) {
        sprintf(error,"Id: Cannot open '%s'", pathname);
        perror(error);
        return -1;
    }
    /* Find load name (no directory info) */
    ldname = strrchr(pathname,'/');
    if (ldname == NULL)
        ldname = fname;
    else
        ldname++;
    IdrUnload(ldname);      /* Unload, if it was previously loaded */
    rc = IdrQuery(fd,&li);  /* Query the storage requirements */
    if (rc != 0) {
        IdrMsg(NULL,NULL,0); /* Print any errors that may have occurred */
    } else {
        rc = IdrLoad(fd,ldname,&li); /* Load file into memory */
        IdrMsg(ldname,NULL,0); /* Display any messages */
        if (li.symbol.size > 0 && rc >= 0) {
            statsym_update(&li); /* Update symbol table */
            IdrResolve(); /* Resolve references to the newly loaded file. */
        }
    }
    close(fd);
    return rc;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **Id()**, p. 352
- **IdrMsg()**, p. 368
- **IdrQuery()**, p. 379

- **ldrResolve()**, p. 370
- **ldrLink()**, p. 361
- **statsym_update()**, p. 782

IdrMoveData()

Synopsis

```
#include <ldrLib.h>

int IdrMoveData( void *objp, void *datap, ldrInfo_t *info );
```

Library

ldrLib.a

Description

IdrMoveData() copies the data from a loaded object image pointed to by *objp* to a new location pointed to by *datap*. The object image must be in XCOFF format for XCOFF-based OS Open or ELF format for ELF-based OS Open. For ELF format the text and data will both be moved.

objp points to an XCOFF or ELF executable object image which has been loaded into memory. If *datap* is NULL, the target location is allocated from heap storage. Otherwise, data is copied to the location specified by *datap*. *info* points to a structure of type **ldrInfo_t** that must be initialized by calling **IdrQimage()** for the object image. The structure will be modified to reflect the new data location. If *info* is NULL it is ignored.

If successful, **IdrMoveData()** returns 0. Otherwise, **IdrMoveData()** returns -1.

Note: The target and source area must not overlap.

Errors

None.

Example

See **IdrLink()**, p. 361.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **IdrLink()**, p. 361
- **IdrQimage()**, p. 369

Synopsis

```
#include <ldrLib.h>

int IdrMoveSym( void *objp, void *symp, ldrInfo_t *info );
```

Library

ldrLib.a

Description

IdrMoveSym() copies parts of the symbol section from a loaded object image pointed to by *objp* to a new location pointed to by *symp*. Only external symbols are actually copied. The object image must be in XCOFF format for XCOFF-based OS Open or ELF format for ELF-based OS Open.

objp points to an XCOFF or ELF executable object image which has been loaded into memory. If *symp* is NULL, the target location is allocated from heap storage. Otherwise, symbols are copied to the location specified by *symp*. *info* points to a structure of type **ldrInfo_t** that must be initialized by calling **IdrQimage()** for the object image. The structure will be modified to reflect the new symbol information. If *info* is NULL it is ignored.

For ELF-based OS Open the symbols cannot be processed if not moved by the **IdrMoveSym()** function. This is true even if the image is loaded into read-write memory.

If successful, **IdrMoveSym()** returns 0. Otherwise, **IdrMoveSym()** returns -1.

Note: The target and source area must not overlap.

Errors

None.

Example

See **IdrLink()**, p. 361.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **IdrLink()**, p. 361
- **IdrQimage()**, p. 369

IdrMsg()

Synopsis

```
#include <ldrLib.h>
int IdrMsg( char *ldname, char *buf, size_t bufsize );
```

Library

ldrLib.a

Description

IdrMsg() displays error and information messages from loader function calls.

Displayed messages include unresolved external references and duplicate symbols.

If *buf* parameter is NULL, output is written to the standard output. Otherwise, output is stored in the buffer pointed to by *buf* until *bufsize* is reached.

If successful, **IdrMsg()** returns 0. Otherwise, **IdrMsg()** returns -1.

Errors

None.

Example

See **IdrLoad()**, p. 363.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **IdrLink()**, p. 361
- **IdrLoad()**, p. 363

Synopsis

```
#include <ldrLib.h>

int ldrQimage( void *objp, ldrInfo_t *info );
```

Library

ldrLib.a

Description

ldrQimage() reads information from a loaded object image. The object image must be in XCOFF format for XCOFF-based OS Open or ELF format for ELF-based OS Open.

objp points to an XCOFF or ELF executable object file which has been loaded into memory. The information is stored in the **ldrInfo_t** structure pointed to by *info*. The format of the structure is defined in **<ldrLib.h>**.

If successful, **ldrQimage()** returns 0. Otherwise, **ldrQimage()** returns -1.

Errors

None.

Example

See **ldrLink()**, p. 361.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **ldrLink()**, p. 361

IdrResolve()

Synopsis

```
#include <ldrLib.h>
int IdrResolve( );
```

Library

ldrLib.a

Description

IdrResolve() resolves references between all loaded objects that have resident loader tables.

IdrResolve() allows two or more objects that refer to each other to be loaded and resolved independently.

If successful, **IdrResolve()** returns 0. Otherwise, **IdrResolve()** returns -1.

Errors

None.

Example

See **IdrLoad()**, p. 363.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **IdrLink()**, p. 361
- **IdrLoad()**, p. 363

Synopsis

```
#include <ldrLib.h>

unsigned long IdrSym_get( char *ldname, char *sname );
```

Library

ldrLib.a

Description

IdrSym_get() returns the value of the symbol *sname* from the loaded object named by *ldname*. **IdrSym_get()** uses the loader section to get the symbol information to work with images that do not contain symbol sections.

IdrSym_get() does not work if **IdrFree()** was previously called to free the loader section.

If **IdrSym_get()** fails, it returns 0.

Errors

None.

Example

The following example links a loaded object file image and calls a function named by the *sname* parameter.

```
/*
 * Link and call a function in a read-only loaded object file image.
 */

int linkcall(char *fname, int arg) {
    void *objp=(void *)0xFFFFF4000;    /* Address of image */
    int rc;
    char *lname="temp.ld";
    int (*funcp)(int);
    ldrInfo_t ldi;

    /* Query object file image */
    rc = ldrQimage(objp, &ldi);
    if (rc != 0) {
        ldrMsg(NULL,NULL,0);
        printf("ldrQimage() error\n");
        return -1;
    }
}
```

IdrSym_get()

```
/* Move data to read-write memory. */
rc = IdrMoveData(objp,NULL,&ldi);
if (rc != 0) {
    IdrMsg(NULL,NULL,0);
    printf("IdrMoveData() error\n");
    return -1;
}

/* Link the object. */
rc = IdrLink(objp,lname,&ldi);
if (rc != 0) {
    IdrMsg(lname,NULL,0);
    if (rc < 0)
        return -1;
}

/* Get the function pointer for "fname". */
funcp = (int (*)(int))(IdrSym_get(lname,fname));
if (!funcp) {
    printf("symbol \"%s\" not found in %s\n",fname,lname);
    return -1;
}

/* Call the function. */
rc = (*funcp)(arg);

/* Unlink the object. */
IdrUnlink(lname);

/* Free storage for memory. */
free(ldi.data.addr);

return rc;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **Id()**, p. 352
- **IdrLink()**, p. 361
- **IdrLoad()**, p. 363

Synopsis

```
#include <ldrLib.h>
int IdrUnlink( char *Idname );
```

Library

ldrLib.a

Description

IdrUnlink() unlinks the object pointed to by *Idname* that was previously linked by **IdrLink()**.

IdrUnlink() frees all memory used by the loaded object, removes the object's external symbols, and unresolves other objects' references to the removed external symbols.

If successful, **IdrUnlink()** returns 0. Otherwise, **IdrUnlink()** returns -1.

Errors

None.

Example

See **IdrSym_get()**, p. 371.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **IdrLink()**, p. 361

ldr_ldinfo()

Synopsis

```
#include <ldrLib.h>

int ldr_ldinfo( void *buffer, size_t buflen );
```

Library

ldrLib.a

Description

ldr_ldinfo() writes information about all loaded objects into the location pointed to by *buffer*. The argument *buflen* specifies the maximum amount of data that can be written to the buffer.

The format of the data written to the buffer, defined by the structure **ldinfo_t** in the file **<ldrLib.h>**, follows:

Field	Length
unsigned long ldinfo_next (offset)	4 bytes
long fd (set to -1)	4 bytes
unsigned long textorig	4 bytes
unsigned long textsize	4 bytes
unsigned long dataorig	4 bytes
unsigned long datasize	4 bytes
char ldname[]	X bytes
char membername ('\0')	1 byte
unsigned long xcoff_hdr	4 bytes
TOTAL	30+X

The data is repeated for each loaded object.

If the value of *ldinfo_next* is 0, no further loader blocks are present. Otherwise *ldinfo_next* contains the offset of the next block in the buffer.

ldname contains no path information and is terminated with a NULL (\0).

xcoff_hdr contains the offset of the text section in the XCOFF file (available only when loading XCOFF files).

The value of the *ldname* of the first entry (base load) is **BASELOAD_LDNAME** as defined in the file **<ldrLib.h>**.

Note: ELF files can only contain one text section and one data section, and these will be loaded into contiguous memory.

If successful, **ldr_ldinfo()** returns the number of bytes written to the buffer. Otherwise, **ldr_ldinfo()** returns -1.

Errors

None.

Example

The following example prints a map of all loaded objects.

```
/* OS Open Map Display */
#include <ldrLib.h>
#include <stdlib.h>

int map() {
    int rc;
    void *buffer;
    int buflen = 0, entnum=1;
    ldinfo_t *n;
    unsigned long *xcoff_hdr_p;

    do {
        buflen += 1024;
        buffer=malloc(buflen);
        if (buffer == NULL)
            return -1;
        rc = ldr_ldinfo(buffer,buflen);
        if ( rc == -1)
            free(buffer);
    } while (rc == -1);
    n = (ldinfo_t *)buffer;
    do {
        printf("Entry %d:\n",entnum++);
        printf("  Object name: %s\n",n->ldname);
        printf("  Text origin: %#x\n",n->textorig);
        printf("  Text length: %#x\n",n->textsize);
        printf("  Data origin: %#x\n",n->dataorig);
        printf("  Data length: %#x\n",n->datasize);
        xcoff_hdr_p = (unsigned long *)((char *)n+XCOFFHOFF+strlen(n->ldname));
        printf("  XCOFF headr: %#x\n",*xcoff_hdr_p);
        printf("\n");
        n = (ldinfo_t *)((char *)buffer+n->ldinfo_next);
    } while (buffer != (void *)n);
    free(buffer);
    return 0;
} /* end map() */
```

Idr_Idinfo()

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **Id()**, p. 352
- **IdrLink()**, p. 361
- **IdrLoad()**, p. 363

Synopsis

```
#include <ldrLib.h>
int IdrFree( char *ldname );
```

Library

ldrLib.a

Description

IdrFree() frees storage used to hold the loader section of the loaded object pointed to by *ldname* and makes the storage available to OS Open. **IdrFree()** fails if the loaded object has unresolved references.

If successful, **IdrFree()** returns 0. Otherwise, **IdrFree()** returns -1.

Note: **IdrFree()** should not be called if external objects that reference the function may be unloaded at a future time.

Errors

None.

Example

See **Id()**, p. 352.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **IdrLoad()**, p. 363

IdrLib_init()

Synopsis

```
#include <IdrLib.h>
int IdrLib_init(void);
```

Library

IdrLib.a

Description

IdrLib_init() initializes the loader library functions.

If unsuccessful, **IdrLib_init()** returns -1. Otherwise, **IdrLib_init()** returns 0.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	No

References

- **Id()**, p. 352
- **IdrUnload()**, p. 381

Synopsis

```
#include <ldrLib.h>
int ldrQuery( int fd, ldrInfo_t *info );
```

Library

ldrLib.a

Description

ldrQuery() reads information from an opened object file.

The information is stored in the **ldrInfo_t** structure pointed to by *info*. The format of the structure is defined in **<ldrLib.h>**.

If successful, **ldrQuery()** returns 0. Otherwise, **ldrQuery()** returns -1.

Errors

None.

Example

See **ldrLoad()**, p. 363.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **ldrLoad()**, p. 363

ldrStatus()

Synopsis

```
#include <ldrLib.h>
int ldrStatus( char *buf, size_t bufsize );
```

Library

ldrLib.a

Description

ldrStatus() displays status information for all loaded objects.

If *buf* is NULL, output is written to the standard output. Otherwise, output is stored in the buffer pointed to by *buf* until *bufsize* is reached.

If successful, **ldrStatus()** returns 0. Otherwise, **ldrStatus()** returns -1.

Errors

None.

Example

See **ldrLoad()**, p. 363.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **ld()**, p. 352
- **ldrLink()**, p. 361
- **ldrLoad()**, p. 363

Synopsis

```
#include <ldrLib.h>
int IdrUnload( char *Idname );
```

Library

ldrLib.a

Description

IdrUnload() unloads the object pointed to by *Idname* that was previously loaded by **Id()** or **IdrLoad()**.

IdrUnload() frees all memory used by the loaded object, remove the object's external symbols, and unresolves other objects' references to the removed external symbols.

If successful, **IdrUnload()** returns 0. Otherwise, **IdrUnload()** returns -1.

Errors

None.

Example

See **IdrLoad()**, p. 363.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **Id()**, p. 352
- **IdrLoad()**, p. 363

library_list()

Synopsis

```
#include <kadtLib.h>
int library_list( char **bufp );
```

Library

kadtLib.a

Description

library_list() provides formatted information about libraries registered using **package_install()**.

The information is part of the **kda_dump()** display.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

library_list() returns 0.

Errors

None.

Example

The following shows a sample listing of libraries:

```
OS OPEN>library_list()
Installed packages  Anchor/ Package name
0x0  shell.o  4.11  8/22/95
0x27f0a8  tnetdLib.a  4.11  8/22/95
0x0  rsldLib.a  4.13  9/21/95
0x15d0b8  symLib.a  4.11  8/22/95
0x15b904  dbLib.a  4.2  9/28/95
0x2796cc  netLib.a  4.11  8/22/95
0x2777d8  tcpipLib.a  4.11  8/22/95
0x15b238  enetLib.a  4.3  9/20/95
0x15b098  fatLib.a  4.17  10/11/95
0x15afc8  dstkLib.a  4.5  9/20/95
0x4b9a9c  fsLib.a  4.11  8/22/95
0x0  ttyLib.a  4.11  8/22/95
0x15ad04  asyncLib.a  4.2  8/2/95
0x158488  devLib.a  4.13  9/26/95
-----
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **kda_dump()**, p. 349
- **package_install()**, p. 484

link()

Synopsis

```
#include <unistd.h>

int link( const char *existing, const char *new );
```

Library

devLib.a

Description

link() creates a directory entry using the name pointed to by *new* and associates the new entry with the file pointed to by *existing*.

File system implementations may limit the number of links that can exist for a file.

If successful, **link()** returns 0. Otherwise, **link()** returns -1 and sets *errno*.

Errors

[EACCES]	Search permission is denied for a component of the path, or the requested link requires writing while write permission is denied.
[EEXIST]	The link pointed to by <i>new</i> already exists.
[EMLINK]	The number of links to the existing file would exceed {LINK_MAX} .
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of the existing or new file name exceeds {PATH_MAX} .
[ENOENT]	<i>existing</i> does not exist, or <i>new</i> points to an empty string.
[ENOSPC]	The directory containing the link could not be extended.
[ENOTDIR]	A component of either path prefix is not a directory.
[EPERM]	The file named by <i>existing</i> is a directory
[EROFS]	Requested link requires writing in a directory on a read-only file system.
[EXDEV]	The requested link is between different file systems.

Example

The following example extracts the number of existing links for a file, compares the number against path configuration limits, and creates another link.

```
#include <unistd.h>
int check_link(char *fname)
{
    int rc;
    long max_links;
    int current_links;
```



```
struct stat stat_buf;

/* obtain file statistics */
rc = stat(fname, &stat_buf);
if (rc != 0) {
    perror("stat call");
    return(rc);
}
current_links = stat_buf.st_nlink;
max_links = pathconf(fname, _PC_LINK_MAX);
if (current_links >= max_links) {
    printf("Limit of %d links reached for file %s\n",
        current_links, fname);
    return(-1);
}
rc = link(fname, "test_link");
if (rc != 0) {
    perror("link call");
    return(rc);
}
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.3.4
- **unlink()**, p. 923

listen()

Synopsis

```
#include <sys/socket.h>
int listen( int s, int backlog );
```

Library

tcpipLib.a

Description

listen() listens on the socket named by *s* for connections and limits the backlog of incoming connections.

For a socket to accept connections, three events must occur. A socket *s* is created using **socket()**. **listen()** establishes the ability to accept incoming connections and specifies a queue limit for the incoming connections. Finally, **accept()** accepts connections.

backlog defines the maximum size of the queue of pending connections. If a connection request arrives when the queue is full, the client may receive an **[ECONNREFUSED]** error. If the underlying protocol supports retransmission, the request may be ignored; a retry may succeed. The maximum queue size is defined as **SOMAXCONN**, which is defined in **<sys/socket.h>**.

Note: **listen()** applies only to sockets of type **SOCK_STREAM**.

If successful, **listen()** returns 0. Otherwise, **listen()** returns -1 and sets *errno*.

Errors

[EBADF]	The descriptor <i>s</i> is not valid.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENOTSOCK]	The descriptor <i>s</i> references a file, not a socket.
[EOPNOTSUPP]	The socket type does not supports listen() .

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.9
- **accept()**, p. 52
- **connect()**, p. 132
- **socket()**, p. 760

Synopsis

```
#include <locale.h>

struct lconv *localeconv( void );
```

Library

cLib.a

Description

localeconv() returns a pointer to the structure **struct lconv**.

The OS Open implementation of **localeconv()** supports only the C locale.

The members of **struct lconv** of type **char *** point to "", except **decimal_point**, which points to ".". The members of type **char** have the value **CHAR_MAX**, which is defined in the file **<limits.h>**. **struct lconv** is defined in the file **<locale.h>**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.4.2.1
- **setlocale()**, p. 723

localtime()

Synopsis

```
#include <time.h>
struct tm *localtime( const time_t *caltime );
```

Library

cLib.a

Description

localtime() converts calendar time to broken-down time, expressed as local time, and returns a pointer to the structure containing the broken-down time.

caltime specifies the calendar time. The resulting broken-down time is stored in a static structure, making this function not async safe.

Errors

None.

Example

asctime(), p. 62

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.12.3.4

Synopsis

```
#include <time.h>

struct tm *localtime_r( const time_t *caltime,
                        struct tm *result );
```

Library

cLib.a

Description

localtime_r() is a reentrant version of **localtime()**.

localtime_r() converts calendar time to broken-down time, expressed as local time, and stores the result in the structure pointed to by *result*.

If successful, **localtime_r()** returns the structure **tm** containing the result.

Errors

None.

Example

See **asctime()**, p. 62.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §19.2.3.2
- **localtime()**, p. 388

log()

Synopsis

```
#include <math.h>
double log( double x );
```

Library

mathLib.a

Description

log() computes the natural logarithm of x .

If x is negative, a domain error occurs and **log()** sets *errno*. If x is 0 or the result is out of range, a range error occurs and **log()** sets *errno*.

Errors

[EDOM]	x is negative.
[ERANGE]	x is 0, or the result is out of range.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.4.4

Synopsis

```
#include <math.h>
double log10( double x );
```

Library

mathLib.a

Description

log10() computes the base-ten logarithm of x .

If x is negative, a domain error occurs and **log10()** sets *errno*. If x is 0 or the result is out of range, a range error occurs and **log10()** sets *errno*.

Errors

[EDOM]	x is negative.
[ERANGE]	x is 0, or the result is out of range.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.4.5

longjmp()

Synopsis

```
#include <setjmp.h>
void longjmp( jmp_buf env, int val );
```

Library

cLib.a

Description

longjmp() restores the environment saved by a call to **setjmp()**.

longjmp() returns *val* to the function that called **setjmp()**, as if the function had just returned.

If *val* is 0, **setjmp()** returns 1.

longjmp() changes the execution context and does not return to its caller.

Errors

None.

Example

See **shm_open()**, p. 734.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.6.2
- **setjmp()**, p. 722

Synopsis

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

off_t lseek( int filedes, off_t offset, int whence );
```

Library

devLib.a

Description

lseek() sets the file offset for the open file descriptor associated with *filedes*.

whence is one of the following constants defined in **<unistd.h>**.

SEEK_SET	File offset set to <i>offset</i> from beginning of the file
SEEK_CUR	File offset set to <i>offset</i> from the file position indicator
SEEK_END	File offset set to <i>offset</i> from end-of-file (EOF)

lseek() allows the file offset to be placed past the end of an existing file. This action does not increase the size of the file unless data is written to the new location. Then all bytes between the original EOF and the new offset are set to 0 until data is written in the gap.

If successful, **lseek()** returns the resulting offset location. Otherwise, **lseek()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>whence</i> or the resulting file offset is not valid.
[EBADF]	<i>filedes</i> is not a valid file descriptor.
[ESPIPE]	<i>filedes</i> is associated with a pipe of FIFO.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1, §6.5.3

m_free()

Synopsis

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

struct mbuf *m_free( struct mbuf *m );
```

Library

tcpipLib.a

Description

m_free() returns the memory buffer (mbuf) pointed to by *m* to the mbuf pool.

m_free() returns the address of the mbuf that followed the freed mbuf in the mbuf chain, or a NULL pointer if no mbuf followed the freed mbuf.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **m_freem()**, p. 395
- **MFREE()**, p. 424

Synopsis

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

int m_freem( struct mbuf *m );
```

Library

tcpipLib.a

Description

m_freem() returns, to the mbuf pool, the memory buffer (mbuf) chain pointed to by *m*.

The returned value is unspecified.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **m_free()**, p. 394
- **MFREE()**, p. 424

M_PREPEND()

Synopsis

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

M_PREPEND( struct mbuf *m, int plen, int how );
```

Library

None. This is a macro defined in `<sys/mbuf.h>`.

Description

M_PREPEND() prepends the amount of storage given by *plen* to the data portion of the memory buffer (mbuf) pointed to by *m*.

If there is sufficient space in front of the existing data portion of the specified mbuf, the requested storage is made available in the existing mbuf. Otherwise, a new mbuf is allocated.

OS Open ignores *how*, which, in some systems, specifies whether to wait if a new mbuf must be allocated.

If the allocation fails, the entire chain is freed and *m* is set to NULL.

Errors

[ENOMEM]	Insufficient memory.
----------	----------------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- `m_prepend()`, p. 397

Synopsis

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

m_prepend( struct mbuf *m, int plen, int how );
```

Library

tcpipLib.a

Description

m_prepend() prepends the amount of storage given by *plen* to the data portion of the memory buffer (mbuf) pointed to by *m*.

If there is sufficient space in front of the existing data portion of the specified mbuf, the requested storage is made available in the existing mbuf. Otherwise, a new mbuf is allocated.

OS Open ignores *how*, which, in some systems, specifies whether to wait if a new mbuf must be allocated.

Errors

[ENOMEM] Insufficient memory.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **M_PREPEND()**, p. 396

malloc()

Synopsis

```
#include <stdlib.h>

void *malloc( size_t num_bytes );
```

Library

cLib.a

Description

malloc() allocates *num_bytes* bytes of memory.

The content of the allocated memory is indeterminate.

If successful, **malloc()** returns a pointer to the allocated memory. Otherwise, **malloc()** returns NULL.

Errors

None.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.3.3
- **calloc()**, p. 89
- **free()**, p. 228
- **realloc()**, p. 638

Synopsis

```
#include <stdlib.h>

int mblen ( const char *s, size_t n );
```

Library

cLib.a

Description

mblen() returns 1, unless *n* equals 0 (**mblen()** returns -1) or *s* is NULL or points to a NULL string (**mblen()** returns 0).

OS Open does not support multibyte characters; **mblen()** is provided for ANSI C compatibility.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.7.1

mbtowc()

Synopsis

```
#include <stdlib.h>

int mbtowc( wchar_t *pwc const char *s, size_t n );
```

Library

cLib.a

Description

mbtowc() returns 1.

OS Open does not support multibyte characters; **mbtowc()** is provided for ANSI C compatibility.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.7.2

Synopsis

```
#include <stdlib.h>
size_t mbstowcs( wchar_t *pcws, const char *s, size_t n );
```

Library

cLib.a

Description

mbstowcs() returns the value of *n*.
OS Open does not support multibyte characters; **mbstowcs()** is provided for ANSI C compatibility.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.8.1

memchr()

Synopsis

```
#include <string.h>

void *memchr( const void *s, int c, size_t n );
```

Library

cLib.a

Description

memchr() locates the first occurrence of *c* (converted to an **unsigned char**) in the first *n* characters (interpreted as **unsigned char**) of the object pointed to by *s*.

memchr() returns a pointer to the located character, or a NULL pointer if *c* does not occur in the object.

Errors

None.

Example

See **memmove()**, p. 415.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.5.1
- **strchr()**, p. 787
- **strrchr()**, p. 803

Synopsis

```
#include <string.h>

int memcmp( const void *s1, const void *s2, size_t n );
```

Library

cLib.a

Description

memcmp() compares the first *n* characters of the object pointed to by *s1* and the first *n* characters of the object pointed to by *s2*.

The **memcmp()** function returns an integer greater than, equal to, or less than 0, accordingly as the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*.

Errors

None.

Example

See **memheap_query()**, p. 410.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.4.1
- **strcmp()**, p. 790
- **strncmp()**, p. 800

memcpy()

Synopsis

```
#include <string.h>

void *memcpy( void *s1, const void *s2, size_t n );
```

Library

cLib.a

Description

memcpy() copies *n* characters from the object pointed by *s2* to the object pointed to by *s1*.

If copying takes place between objects that overlap, the behavior is undefined. **memmove()** can handle overlapping memory segments.

memcpy() returns the value of *s1*.

Errors

None.

Example

See **shm_open()**, p. 734.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.2.1
- **memmove()**, p. 415
- **strcpy()**, p. 792
- **strncpy()**, p. 801

Synopsis

```
#include <stddef.h>
#include <memLib.h>

void *memheap_alloc( int heap_id, size_t num_bytes );
```

Library

rtxLib.a

Description

memheap_alloc() allocates *num_bytes* bytes from the heap *heap_id*.

The content of the allocated memory is indeterminate.

For applications, the only valid value for *heap_id* is **APPL_HEAP**. When the kernel runs in a single heap configuration, all **APPL_HEAP** requests apply to the single heap.

If successful, **memheap_alloc()** returns a pointer to the allocated memory. Otherwise, **memheap_alloc()** returns NULL and sets *errno*.

Note: For applications, **malloc()** is recommended for its simplicity and portability in ANSI C applications.

Errors

[EINVAL]	<i>heap_id</i> is not an active memory region, KERNEL_HEAP is specified, or <i>num_bytes</i> is 0.
[ENOMEM]	Insufficient heap space.

Errors

See **int_install()**, p. 327.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **memheap_alloc_pages()**, p. 407
- **memheap_free()**, p. 409

memheap_alloc_aligned()

Synopsis

```
#include <stddef.h>
#include <memLib.h>

void *memheap_alloc_aligned( int heap_id, size_t mem_size,
                           unsigned int alignment );
```

Library

rtxLib.a

Description

memheap_alloc_aligned() allocates memory from the heap *heap_id* satisfying the boundary given by *alignment*.

The value of *alignment* must be a power of two. For example, if *alignment* is 8, the allocated memory block is aligned on a 256-byte boundary and its address has at least 8 low-order zero bits.

The content of the allocated memory is indeterminate.

For applications, the only valid value for *heap_id* is **APPL_HEAP**. When the kernel runs in a single heap configuration, all **APPL_HEAP** requests apply to the single heap.

If successful, **memheap_alloc_aligned()** returns a pointer to the allocated memory. Otherwise, **memheap_alloc_aligned()** returns NULL and sets *errno*.

Errors

[EINVAL]	<i>heap_id</i> is not an active memory region, KERNEL_HEAP is specified, or <i>mem_size</i> is 0.
[ENOMEM]	Insufficient heap space to meet alignment constraints.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **memheap_alloc()**, p. 405
- **memheap_free()**, p. 409

Synopsis

```
#include <stddef.h>
#include <memLib.h>

void *memheap_alloc_pages( int heap_id, size_t mem_size );
```

Library

rtxLib.a

Description

memheap_alloc_pages() allocates page-aligned memory from the heap *heap_id*.

The byte value *mem_size* is rounded up to the nearest integral multiple of a page size.

The content of the allocated memory is indeterminate.

For applications, the only valid value for *heap_id* is **APPL_HEAP**. When the kernel runs in a single heap configuration, all **APPL_HEAP** requests apply to the single heap.

If successful, **memheap_alloc_pages()** returns a pointer to the allocated memory. Otherwise, **memheap_alloc_pages()** returns NULL and sets *errno*. For purposes of this function, a page is 4096 bytes.

Errors

[EINVAL]	<i>heap_id</i> is not an active memory region, KERNEL_HEAP is specified, or <i>mem_size</i> is 0.
[ENOMEM]	Insufficient heap space.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **memheap_alloc()**, p. 405
- **memheap_free()**, p. 409

memheap_extend()

Synopsis

```
#include <stddef.h>
#include <memLib.h>

int memheap_extend( int heap_id, void *start_addr,
                    size_t extent );
```

Library

rtxLib.a

Description

memheap_extend() adds a new segment to the heap named by *heap_id* and makes its free space available for allocation.

The segment is defined as beginning with an address pointed to by *start_addr* for *extent* bytes. Since control information is allocated from the supplied memory, *extent* must be at least as large as **MIN_SEGMENT**, a constant defined in **<memLib.h>**.

The value of *heap_id* must be either **KERNEL_HEAP** or **APPL_HEAP**. The specified heap must have been created at OS Open initialization.

memheap_extend() cannot create **APPL_HEAP** because OS Open runs in a single heap configuration. All **APPL_HEAP** requests, including this one, apply to the single heap.

If successful, **memheap_extend()** returns 0. Otherwise, **memheap_extend()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>heap_id</i> is not an active memory heap, or <i>extent</i> is smaller than the minimum segment size.
----------	---

Example

See **getchar_unlocked()**, p. 258.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <memLib.h>

int memheap_free( void *mem_address );
```

Library

rtxLib.a

Description

memheap_free() frees a block of memory previously allocated using one of:

- **memheap_alloc_aligned()**
- **memheap_alloc_pages()**
- **memheap_realloc()**
- **memheap_alloc()**

mem_address points to the address of the block.

If *mem_address* does not refer to a memory area obtained from the heap, the results are indeterminate. However, **memheap_free()** performs internal checks that prevent heap errors in many cases.

If successful, **memheap_free()** returns 0. Otherwise, **memheap_free()** returns -1 and sets *errno*.

If *mem_address* is NULL, **memheap_free()** returns 0.

Errors

[EINVAL]	<i>mem_address</i> does not point to a previously allocated block.
----------	--

Example

See **int_install()**, p. 327.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

memheap_query()

Synopsis

```
#include <memLib.h>

int  memheap_query( int heap_id, memheap_status_t *hp_stat);

typedef struct
{
    long area_size;      /* total heap size */
    long used_space;     /* space in use   */
    long big_block;      /* largest free block */
    long high_water;     /* max space in use */
} memheap_status_t;
```

Library

rtxLib.a

Description

memheap_query() returns the status of the heap *heap_id*.

An OS Open system has at least one heap, **KERNEL_HEAP**, used by the kernel to store internal control blocks. At initialization, an application designer can define another heap called **APPL_HEAP**.

If defined, **APPL_HEAP** satisfies memory requests from **malloc()**, **calloc()**, and **realloc()**, along with memory requests from **memheap_alloc()**, **memheap_alloc_pages()**, and **memheap_alloc_pages()**, that specify **APPL_HEAP**. Otherwise, **KERNEL_HEAP** fulfills memory requests.

If *heap_id* is set to **APPL_HEAP** and the kernel is running in a single heap configuration, **memheap_query()** fails.

Note: The value of *area_size* refers to all available space, not including control information overhead. The value of *used_space* is the sum of the sizes of the blocks marked in use. This value may be greater than the sum of application requests to this heap; allocated blocks are rounded up to meet minimum size and alignment restrictions.

After the value of *high_water* is returned, it is internally reset to the value of *used_space*. This action allows multiple calls to **memheap_query()** to indicate usage patterns over time.

If successful, **memheap_query()** returns 0. Otherwise, **memheap_query()** returns -1 and sets *errno*.

Errors

[EINVAL] *heap_id* is not an active memory heap.

Example

The following example monitors kernel heap memory allocation and logs the information in a file.

The function queries the heap every 5 seconds. If heap allocation changes, a log and time stamp are written to a file. When the heap is running out of memory, the signal **SIGUSR2** is raised, the file is closed, and the function returns.

```
#include <time.h>
#include <memLib.h>
#include <string.h>
#include <stdio.h>
#include <stddef.h>
#include <signal.h>
#include <unistd.h>
#include <pthread.h>

int *oldtype
void *mem_usage(void *file_name)
{
    FILE *file;
    memheap_status_t old;
    memheap_status_t new;
    time_t time_stamp;
    /* Open the given file for writing */
    /* The previous contents are destroyed */
    file = fopen((char *)file_name, "w");
    if(file == NULL)
    {
        raise(SIGINT);
        return;
    }
    /* Set interrupt types to asynchronous */
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, oldtype);
    /* start querying the kernel heap */
    memset((void *)&old, '\0', sizeof(old));
    while(1)
    {
        memheap_query(KERNEL_HEAP, &new);
        /* heap usage has changed */
        if(memcmp((void *)&old, (void *)&new,
            sizeof(memheap_status_t)) != 0)
        {
            /* Log an entry in the file */
            time(&time_stamp);
            fprintf(file, "%s", ctime(&time_stamp));
        }
    }
}
```

memheap_query()

```
    fprintf(file, "Total kernel heap size: %d\n",
            new.area_size);
    fprintf(file, "Bytes currently allocated for use: %d\n",
            new.used_space);
    fprintf(file, "Largest block of free space: %d\n",
            new.big_block);
    fprintf(file, "High water mark: %d\n", new.high_water);
}
/* check to see if there is less than 64 bytes left in heap */
if(new.area_size - new.used_space < 64)
{
    raise(SIGUSR2);
    break;
}
old = new;
sleep(5);
}
/* close file */
fclose(file);
return((void *)&time_stamp);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <stddef.h>
#include <memLib.h>

void *memheap_realloc( void *mem_address, size_t newsize );
```

Library

rtxLib.a

Description

memheap_realloc() changes the size of an allocated block of memory to a size given by *newsize*.

mem_address points to the address of the block to change. The pointer must be NULL or have the value returned by an previous call to one of:

- memheap_alloc_aligned()
- memheap_alloc()
- memheap_alloc_pages()
- memheap_realloc()

If *mem_address* is NULL, **mem_address** behaves like **memheap_alloc()** and allocates a new block of memory of size *newsize*. The return pointer of the altered block may be different, but, up to the smaller of the original size and the new size, the contents of the block are guaranteed to be unchanged.

Note: **memheap_realloc()** always extends memory in the same manner, regardless of the original method of allocation. For example, if the memory to be extended was allocated using **memheap_alloc_pages()**, the new memory will generally not be an integral number of pages aligned on a page boundary.

If the allocation fails, *mem_address* is unchanged and **memheap_realloc** returns NULL and sets *errno*.

Errors

[EINVAL]	The value of <i>mem_address</i> or <i>newsize</i> is not valid.
[ENOMEM]	Insufficient space in heap.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **memheap_alloc()**, p. 405

memheap_replace()

Synopsis

```
#include <memLib.h>
```

```
int memheap_replace( heap_func_t *new_func, heap_func_t *old_func );
```

Library

rtxLib.a

Description

memheap_replace() replaces OS Open memory management functions with functions supplied by the user. User memory management functions are specified in the *new_func* structure. OS Open memory management functions are placed in the *old_func* structure if the *old_func* variable is not NULL.

All OS Open memory management requests are handled by the new functions after replacement. OS Open memory management functions can be placed back by filling the *new_func* structure with NULL function pointers.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **memheap_alloc()**, p. 405
- **memheap_alloc_aligned()**, p. 406
- **memheap_alloc_pages()**, p. 407
- **memheap_extend()**, p. 408
- **memheap_free()**, p. 409
- **memheap_query()**, p. 410
- **memheap_realloc()**, p. 413

Synopsis

```
#include <string.h>

void *memmove( void *s1, const void *s2, size_t n );
```

Library

cLib.a

Description

memmove() copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*.

Copying takes place as if *n* characters from the object pointed to by *s2* are copied into a temporary array of *n* characters that does not overlap the objects pointed to by *s1* and *s2*, and are then copied into the object pointed to by *s1*.

memmove() returns the value of *s1*.

Errors

None.

Example

The following example removes all instances of a character in a block of memory by moving the data following the character one space to the right. A space character (' ') is appended to the block of memory for each character removed.

```
#include <stddef.h>
#include <string.h>

void remove_char(int ch, int num, void *beg)
{
    int count = 0;
    int number = num;
    void *current = beg;
    /* Remove each occurrence of the character */
    while((current = memchr(current, ch, number)) != NULL)
    {
        number = num - ((char *)current - (char *)beg);
        memmove(current, (void *)((char *)current + 1), number);
        count++;
    }
    /* Append the space character at the end of the memory block */
    current = (char *)beg + (num - count);
    memset(current, ' ', count);
}
```

memmove()

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.2.2
- **memcpy()**, p. 404
- **strcpy()**, p. 792
- **strncpy()**, p. 801

Synopsis

```
#include <memLib.h>
void *mempool_alloc( Pool_ID id );
```

Library

rtxLib.a

Description

mempool_alloc() allocates a block of memory from the buffer pool *id*.

The content of the returned memory is indeterminate.

If successful, **mempool_alloc()** returns a pointer to the allocated block of memory. Otherwise, **mempool_alloc()** returns NULL and sets *errno*.

Errors

[EINVAL]	<i>id</i> is not an active buffer pool.
[ENOMEM]	All blocks in the pool are allocated.

Example

See **mempool_init()**, p. 420.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **mempool_free()**, p. 419

mempool_destroy()

Synopsis

```
#include <memLib.h>
int mempool_destroy( Pool_ID id );
```

Library

rtxLib.a

Description

mempool_destroy() removes from use the buffer pool specified by *id*.

mempool_destroy() invalidates subsequent operations against the buffer pool and frees internal control blocks but does not modify the pool memory.

Note: **mempool_destroy()** unconditionally destroys the buffer pool. Applications can use **mempool_query()** to determine if any blocks are in use before calling **mempool_destroy()**.

If successful, **mempool_destroy()** returns 0. Otherwise, **mempool_destroy()** returns `-1` and sets *errno*.

Errors

[EINVAL] *id* is not an active buffer pool.

Example

See **mempool_init()**, p. 420.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **mempool_init()**, p. 420

Synopsis

```
#include <memLib.h>

int mempool_free( Pool_ID id, void *block_pointer );
```

Library

rtxLib.a

Description

mempool_free() returns a block of memory pointed to by *block_pointer* to the buffer pool specified by *id*.

The block must have been allocated by a previous call to **mempool_alloc()**. The block is checked to ensure that it belongs to *id* and was not already returned.

If successful, **mempool_free()** returns 0. Otherwise, **mempool_free()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>id</i> is not an active buffer pool, <i>block_pointer</i> points to a block that does not belong to this pool, or the block was already returned to the buffer pool.
----------	---

Example

See **mempool_init()**, p. 420.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **mempool_alloc()**, p. 417

mempool_init()

Synopsis

```
#include <stddef.h>
#include <memLib.h>

int mempool_init( Pool_ID *id, void *start_addr,
                 size_t extent, size_t b_size );
```

Library

rtxLib.a

Description

mempool_init() creates a buffer pool in the range of memory specified by *start_addr* and *extent* and stores its pool ID into *id*.

The buffer pool is divided into an integral number of blocks, each *b_size* bytes long. If *b_size* is less than 16, 16 is used.

If successful, **mempool_init()** returns the number of blocks in the new buffer pool. Otherwise, **mempool_init()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>extent</i> is less than <i>b_size</i> , or <i>extent</i> is 0, or <i>start_addr</i> is NULL.
[ENOMEM]	Insufficient memory is available to allocate control areas.

Example

The following example returns the number of blocks in use for a global buffer pool. If the pool does not exist, a buffer pool is created and three blocks are allocated and assigned to three global variables.

```
#include <memLib.h>
#include <pthread.h>
#include <stddef.h>

#define START_ADDRESS 0x40000
#define ADDRESS_SIZE 2048
#define BLOCK_SIZE 64

Pool_ID global_pool = 0;
void *block1 = NULL, *block2 = NULL, *block3 = NULL;
static pthread_once_t pool_init_control = PTHREAD_ONCE_INIT;
void pool_init(void);

int used_blocks(void)
{
```

```
mempool_status_t pool_status;
int rv;
/* Initialize pool if it does not exist using the pthread_once */
/* function */
pthread_once(&pool_init_control, pool_init);
/* Query pool status to find number of blocks in use */
rv = mempool_query(global_pool, &pool_status);
if(rv < 0)
    return(-1);
/* return number of blocks in use */
return(pool_status.blocks_in_use);
}

void pool_init(void)
{
    /* Create the global memory pool */
    if(mempool_init(&global_pool, (void *)START_ADDRESS,
        size_t)ADDRESS_SIZE, (size_t) BLOCK_SIZE) < 0)
        return;
    /* Allocate three blocks from the pool */
    block1 = mempool_alloc(global_pool);
    block2 = mempool_alloc(global_pool);
    block3 = mempool_alloc(global_pool);
    /* If any of the allocates failed, free any allocated blocks and */
    /* destroy the pool */
    if((block1 == NULL) || (block2 == NULL) || (block3 == NULL))
    {
        if(block1 != NULL)
            mempool_free(global_pool, block1);
        if(block2 != NULL)
            mempool_free(global_pool, block2);
        if(block3 != NULL)
            mempool_free(global_pool, block3);
        mempool_destroy(global_pool);
        block1 = block2 = block3 = NULL;
        global_pool = 0;
    }
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **mempool_destroy()**, p. 418

mempool_query()

Synopsis

```
#include <memLib.h>

int mempool_query( Pool_ID id, struct mempool_status_t *p_status );

typedef struct
{
    int blocks_in_use; /* blocks in use */
    int blocks_free;   /* free blocks  */
    int block_size;    /* partition block size */
    int high_water;    /* maximum number of blocks in use */
} mempool_status_t;
```

Library

rtxLib.a

Description

mempool_query() returns pool statistics into the structure pointed to by *p_status*.

After the value of *high_water* is returned, it is internally reset to the value of *blocks_in_use*. This action allows multiple calls to **mempool_query()** to determine storage usage patterns over time.

If successful, **mempool_query()** returns 0. Otherwise, **mempool_query()** returns -1 is returned and sets *errno*.

Errors

[EINVAL] *id* is not an active buffer pool.

Example

See **mempool_init()**, p. 420.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <string.h>

void *memset( void *s, int c, size_t n );
```

Library

cLib.a

Description

memset() copies the value of *c* (converted to an **unsigned char**) into the first *n* characters of the object pointed to by *s*.

memset() returns the value of *s*.

Errors

None.

Example

See **flockfile()**, p. 205.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.6.1

MFREE()

Synopsis

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

MFREE( struct mbuf *m, struct mbuf *n );
```

Library

None. This is a macro defined in `<sys/mbuf.h>`.

Description

MFREE() returns the memory buffer (mbuf) pointed to by *m* to the mbuf pool and places the address of the next mbuf in the mbuf chain, if present, in *n*.

If no mbuf follows the mbuf pointed to by *m*, *n* is set to NULL.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- `m_free()`, p. 394
- `m_freem()`, p. 395

Synopsis

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

MGET( struct mbuf *m, int how, int type );
```

Library

None. This is a macro defined in `<sys/mbuf.h>`.

Description

MGET() allocates a memory buffer (mbuf) and stores its address in the object pointed to by *m*.

OS Open ignores the parameter *how*, which in some systems specifies whether to wait if no memory for a buffer is immediately available.

The type of the returned mbuf is set to *type*.

If unsuccessful, **MGET()** sets *m* to NULL.

Errors

[ENOMEM]	Insufficient memory.
----------	----------------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **MGETHDR()**, p. 426

MGETHDR()

Synopsis

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

MGETHDR( struct mbuf *m, int how, int type );
```

Library

None. This is a macro defined in `<sys/mbuf.h>`.

Description

MGETHDR() allocates the first (header) memory buffer (mbuf) in an mbuf chain and stores the address of the mbuf in the object pointed to by *m*.

Because the header mbuf contains control information, it has a smaller data portion than other mbufs.

The OS Open implementation of **MGETHDR()** ignores *how* and *type*.

If unsuccessful, **MGETHDR()** sets *m* to NULL.

Errors

[ENOMEM]	Insufficient memory.
----------	----------------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **MGET()**, p. 425

Synopsis

```
#include <types.h>
#include <sys/stat.h>
int mkdir( const char *path, mode_t mode );
```

Library

devLib.a

Description

mkdir() creates an empty directory with the name pointed to by *path*.

Directory file permission bits are initialized from *mode*.

If successful, **mkdir()** returns 0. Otherwise, **mkdir()** returns -1 and sets *errno*.

Errors

[EACCES]	Search permission is denied for a component of the path, or the requested operation requires writing while write permission is denied.
[EEXIST]	The named file already exists.
[EMLINK]	The number of links to the parent directory would exceed {LINK_MAX} .
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of the existing or new file names exceeds {PATH_MAX} .
[ENOENT]	A component of the path prefix does not exist.
[ENOSPC]	The file system has insufficient space to contain the new directory or to extend the parent directory.
[ENOTDIR]	A component of <i>path</i> is not a directory.
[EPERM]	The file named by <i>path</i> is a directory.
[EROFS]	The parent directory is on a read-only file system.

Example

The following example creates a directory and a file within the new directory, queries a path configuration variable, changes the mode of the file, deletes the file, and finally removes the directory.

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

int dir_tst(char *dirname)
{
```

mkdir()

```
int rc;

/* make directory path relative to current one */
rc = mkdir("newdir", S_IRUSR | S_IWUSR);
if (rc != 0) {
    perror("mkdir call");
    return(rc);
}
fd = open("newdir/newfile", O_RDWR | O_CREAT); /* create file */
if (fd == -1) {
    perror("open call");
    return(-1);
}
name_size = fpathconf(fd, _PC_NAME_MAX); /* get name size */
printf("File name size limit for this filesystem is %d characters\n",
    name_size);
close(fd); /* close file descriptor */
rc = chmod("newdir/newfile", S_IRUSR); /* change to read only */
if (rc != 0) {
    perror("chmod call");
    return(-1);
}
unlink("newdir/newfile"); /* unlink file */
rc = rmdir("newdir");
if (rc != 0) {
    perror("rmdir call");
    return(rc);
}
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.4.1
- **rmdir()**, p. 654

Synopsis

```
#include <time.h>
time_t mktime( struct tm *time );
```

Library

cLib.a

Description

mktime() converts local time, stored as a structure pointed to by *time*, into a **time_t** type suitable for use with other time functions.

mktime() is the inverse of **gmtime()**.

If successful, **mktime()** returns the calendar time type **time_t**. If the calendar time cannot be represented, **mktime()** returns the value **(time_t)(-1)**.

Errors

None.

Example

See **scanf()**, p. 681.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.12.2.3
- **ctime()**, p. 149
- **gmtime()**, p. 299
- **localtime()**, p. 388
- **time()**, p. 886

mlock()

Synopsis

```
#include <sys/mman.h>
int mlock( void *addr, size_t len );
```

Library

devLib.a

Description

mlock() causes all pages mapped by OS Open in the range given by the address pointed to by *addr* for *len* bytes to be memory resident until unlocked.

Because OS Open does not swap real-memory pages to secondary storage, this function is not strictly necessary, but it is included for compatibility with POSIX application profiles. It only checks parameters for validity. The object pointed to by *addr* must be on a page boundary.

If successful, **mlock()** returns 0. Otherwise, **mlock()** returns -1 and sets *errno*.

Errors

[EINVAL]	The object pointed to by <i>addr</i> is not on a page boundary.
----------	---

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §12.1.2
- **munlock()**, p. 459

Synopsis

```
#include <sys/mman.h>
int mlockall( int flags );
```

Library

devLib.a

Description

mlockall() causes all pages mapped by OS Open to be memory resident until unlocked.

Because OS Open does not swap real-memory pages to secondary storage, this function is not strictly necessary, but it is included for compatibility with POSIX application profiles. **mlockall()** only checks parameters for validity.

flags is constructed from one of the following:

MCL_CURRENT	Lock all currently mapped pages.
MCL_FUTURE	Lock all pages requested in the future.

If successful, **mlockall()** returns 0. Otherwise, **mlockall()** returns -1 and sets *errno*.

Errors

[EINVAL] *flags* is not **MCL_CURRENT** or **MCL_FUTURE**.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §12.1.1
- **munlockall()**, p. 460

mmap()

Synopsis

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

void *mmap( void *addr, size_t len, int prot, int flags, int fildes,
            off_t offset );
```

Library

devLib.a

Description

mmap() establishes a mapping between the OS Open address space and a memory object.

mmap() maps the memory object *fildes* at *offset* for *len* bytes and returns the address of the mapped object in the OS Open address space.

If *addr* is not null, and **MAP_FIXED** is not specified, OS Open attempts to map the requested memory object at the location specified by *addr*. If this is not possible, OS Open returns a suitable mapping.

offset must be a multiple of the OS Open page size.

If unsuccessful, **mmap()** returns **MAP_FAILED** (a constant defined in **<sys/mman.h>**) and sets *errno*.

Note: Because OS Open without virtual memory is a single-process real memory system, without memory protection, a subset of the **mmap()** capabilities defined in POSIX is available.

prot specifies memory protection. It consists of the following flags:

[PROT_READ]	Data can be read.
[PROT_WRITE]	Data can be written.
[PROT_EXEC]	Data can be executed.
[PROT_NONE]	Data cannot be accessed.

Note: For OS Open (without virtual memory), at least **PROT_WRITE** must be specified and the file referenced by *fildes* must be opened **O_RDWR**.

flags provides more information about the handling of mapped data:

[MAP_SHARED]	Share changes.
[MAP_PRIVATE]	Changes are private.
[MAP_FIXED]	Interpret the address pointed to by <i>addr</i> exactly.

Note: For OS Open (without virtual memory), which does not support private mapping or fixed mapping, **MAP_SHARED** must be specified and is the only valid value for this parameter. Ordinarily, this address is used by OS Open to compute a mapping address. OS Open (without virtual memory) ignores the address.

Errors

[EACCES]	<i>fildev</i> does not refer to a file opened read/write.
[EBADF]	<i>fildev</i> is not a valid file descriptor.
[EINVAL]	<i>offset</i> is not a multiple of the OS Open page size, <i>flags</i> is not MAP_SHARED , or <i>prot</i> does not contain PROT_WRITE .
[ENODEV]	<i>fildev</i> refers to an object that cannot be mapped, such as a terminal.
[ENOMEM]	There is insufficient address space available to map the object.
[ENOTSUP]	MAP_FIXED or MAP_PRIVATE was specified for OS Open (without virtual memory).
[ENXIO]	The addresses starting at <i>offset</i> and continuing for <i>len</i> are not valid for the object named by <i>fildev</i> .

Example

See **shm_open()**, p. 734.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

References

- IEEE Draft Standard 1003.4/D13, §12.2.1
- **munmap()**, p. 461

modf()

Synopsis

```
#include <math.h>

double modf( double value, double *iptr );
```

Library

mathLib.a

Description

modf() breaks *value* into integral and fractional parts, returns the fractional part, and stores the integral part in the object pointed to by *iptr*.

The integral and fractional parts have the same sign as *value*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.4.6

Synopsis

```
#include <mqueue.h>
int mq_close( mqd_t mqdes );
```

Library

rtxLib.a

Description

mq_close() removes the association between the message queue descriptor *mqdes* and a message queue.

Using *mqdes* after an **mq_close()** and before an **mq_open()** call produces erroneous results.

If *mqdes* is the only existing descriptor for a message queue, any messages left in the queue are discarded.

If successful, **mq_close()** returns 0. Otherwise, **mq_close()** returns `-1` and sets *errno*.

Errors

[EBADF] *mqdes* is not a valid message queue descriptor.

Example

See **mq_send()**, p. 449.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13 §15.2.2
- **mq_open()**, p. 441
- **mq_unlink()**, p. 457

mq_dump()

Synopsis

```
#include <kadtLib.h>
int mq_dump( mqd_t mqid, char *buf );
```

Library

kadtLib.a

Description

mq_dump() provides information about the message queue *mqid*.

If *buf* is NULL, the output is generated by **bprintf()**. Otherwise, the information is stored in the buffer pointed to by *buf*.

mq_dump() returns -1 if *mqid* is not a valid message queue. Otherwise, **mq_dump()** returns 0.

Errors

None.

Example

The following shows a sample dump for message queue 0x469000.

```
OS OPEN>mq_dump(0x469000)
-----
Msg queue ID: 0x469000
Name: _DBBRKPT_MESSAGE_QUEUE_
Max Msg: 128  Flags: BLOCK
Cur Msg: 0  Msg size: 291
State: NO DESTROY  Notify Thread: 0x468c58 (rsld)  Signal: USR1
Threads waiting to send:
Threads waiting to receive:
-----
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84

Synopsis

```
#include <mqueue.h>

int mq_getattr( mqd_t mqdes, mq_attr_t *mqstat );
```

Library

rtxLib.a

Description

mq_getattr() retrieves the message queue attribute from the message queue *mqdes*.

On return, the following members in the **mq_attr_t** structure pointed to by *mqstat* have the values that were set when the message queue was created or modified by **mq_setattr()**.

The following attributes are set at message queue creation:

mq_maxmsg	Maximum number of outstanding messages on the queue
mq_msgsize	Maximum message size for the queue

The following attribute reflects the status of the message queue:

mq_curmsg	Number of messages in the queue.
-----------	----------------------------------

If successful, **mq_getattr()** returns 0. Otherwise, **mq_getattr()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>mqdes</i> is not a valid message queue descriptor.
---------	---

Example

See **mq_receive()**, p. 446.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13 §15.2.8
- **mq_open()**, p. 441
- **mq_send()**, p. 449

mq_list()

Synopsis

```
#include <kadtLib.h>
int mq_list( char **bufp );
```

Library

kadtLib.a

Description

mq_list() provides formatted information about message queues.

The information is part of the **kda_dump()** display.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

mq_list() returns 0.

Errors

None.

Example

The following shows a sample message queue listing.

```
OS OPEN>mq_list()
Msg queue   Name                                     Cur.Msg
0x469000    _DBBRKPT_MESSAGE_QUEUE_                 0
----- Total Number of MSG Queues: 1 -----
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84
- **kda_dump()**, p. 349

Synopsis

```
#include <mqueue.h>
#include <signal.h>

int mq_notify( mqd_t mqdes, const struct sigevent *evp );
```

Library

rtxLib.a

Description

mq_notify() associates a notification request with the message queue descriptor *mqdes*.

When the message queue changes from empty to non-empty and no threads are waiting to receive a message, the signal specified by the **sigev_signo** field of the **sigevent** structure pointed to by *evp* is sent to the thread requesting notification. If a notification request is pending on the message queue, any attempt to attach another notification request fails.

If *evp* is NULL and a notification request had been associated with the message queue, that notification request is detached and the message queue is available for another notification request.

If any threads are blocked waiting to receive a message from a queue that has an associated notification request and a message is sent to the queue, the thread blocked waiting to receive a message unblocks and receives the message. The signal for the notification request is not sent.

A notification request remains pending on the message queue until it has been removed by another **mq_notify()** function call or the message queue has been destroyed.

Upon successful completion, **mq_notify()** returns 0. Otherwise, **mq_notify()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>mqdes</i> is not a valid message queue descriptor.
[EBUSY]	A notification request was attached to the message queue.
[EINVAL]	<i>evp</i> is not a valid pointer.

Example

See **mq_open()**, p. 441.

mq_notify()

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4/D13 §15.2.6
- **mq_open()**, p. 441
- **mq_send()**, p. 449

Synopsis

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

mqd_t mq_open( const char *name, int oflag, mode_t mode,
               mq_attr_t *attr_to_set );
```

Library

rtxLib.a

Description

mq_open() creates a message queue descriptor that establishes a connection between the system process and a message queue.

Other functions can use the message queue descriptor to refer to the message queue. *name* points to a string naming a message queue. The name space for message queues is managed independently of the name space for files and semaphores. The path name pointed to by *name* should contain fewer than **[NAME_MAX]** characters from the portable file name character set. The name may include slash (/) characters, but they have no special significance. Threads calling **mq_open()** using the same *name* value refer to the same message queue. If the path name does not name an existing message queue and creation is not requested, **mq_open()** fails and returns an error.

oflag requests the desired receive and send access to the message queue.

oflag contains bitwise inclusive OR of values from the following list. Applications should specify one of the first three access modes in *oflag*.

[O_RDONLY]	Open the message queue for receiving messages. The thread can use the returned message queue descriptor with mq_receive() , but not with mq_send() . A message queue may be open for receiving messages multiple times in the same or different threads.
[O_WRONLY]	Open the queue for sending messages. The thread can use the returned message queue descriptor with mq_send() , but not with mq_receive() . A message queue may be open for sending messages multiple times in the same or different threads.
[O_RDWR]	Open the queue for both receiving and sending messages. The thread can use any of the functions allowed for O_RDONLY and O_WRONLY . A message queue may be open for sending messages multiple times in the same or different threads.

Any combination of the remaining flags may be specified in *oflag*:

mq_open()

[O_CREAT]	<p>This option creates a message queue. The option requires two additional arguments: <i>mode</i>, which is ignored, and <i>attr_to_set</i>, which points to a mq_attr structure.</p> <p>If the path name pointed to by <i>name</i> was used to create an message queue that is open, this flag has no effect, except as noted under O_EXCL. Otherwise, an empty message queue is created.</p> <p>If <i>attr_to_set</i> is NULL, the message queue is created using the system default message queue attributes. If <i>attr_to_set</i> is not NULL, the message queue attributes mq_maxmsg and mq_msgsize are set to the values of the corresponding members in the mq_attr structure when a message queue is created, but not on subsequent mq_open() calls.</p>
[O_EXCL]	<p>If O_EXCL and O_CREAT are set and the message queue exists, mq_open() fails. The check for the existence of the message queue and the creation of the message queue (if it does not exist) is atomic with respect to other threads executing mq_open() naming the same <i>name</i> with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, mq_open() fails and sets <i>errno</i> to [EINVAL].</p>
[O_NONBLOCK]	<p>The setting of this flag is associated with the open message queue descriptor and determines whether mq_send() or mq_receive() waits for unavailable resources or messages, or fails with <i>errno</i> set to [EAGAIN]. See mq_receive() on page 446 for details.</p>

Note: Each message descriptor obtained from **mq_open()** might have different attributes. Those attributes include: **O_RDONLY**, **O_WRONLY**, **O_RDWR** and **O_NONBLOCK**.

mq_open() does not add or remove messages from the queue.

If **mq_open** fails, it returns the value (**mqd_t**)−1 and sets *errno*.

Errors

[EEXIST]	O_CREAT and O_EXCL are set, and a message queue associated with <i>name</i> already exists.
[EINVAL]	O_EXCL is set and O_CREAT is not set, or <i>name</i> contains characters not in the portable file name character set.
[ENAMETOOLONG]	The length of <i>name</i> exceeds {NAME_MAX} .

[ENOSPC]	The value of one of the queue attributes would cause some system resource limit to be exceeded.
[ENOENT]	O_CREAT is not set and the named message queue does not exist.

Example

The following example is a main function that creates two messages queues and then creates two threads to write to and read from the queues.

```
#include <mqueue.h>
#include <pthread.h>
#include <cLib.h>
#include <signal.h>
#include <fcntl.h>
#include <config.h>

int signal;
void *sendmsg(void *arg); /* prototype of example under mq_send() */
void *recvmsg(void *arg); /* prototype of example under mq_receive() */
void main();
void panic();
const conf_t _Kernel_Config_block = {
    main,                /* Main function */
    panic,               /* Panic function */
    0x00000000,          /* Main argc */
    0x00040000,          /* Main stack size */
    (void *)0x00000000,  /* Memory heap 1 start address */
    0x00500000,          /* Memory heap 1 size */
    0x00000000,          /* Memory heap 2 start address */
    0x00000000,          /* Memory heap 2 size */
    0x000004b0,          /* Trace table size */
    0x00fe5100,          /* Tick rate */
    0x0fe51000,          /* Round robin time slice */
};

void main()
{
    mq_attr_t mqa;
    mqd_t mq1, mq2;
    pthread_attr_t attr;
    pthread_t t1, t2;
    sigset_t sigs;
    int rv, *status;
    struct sigevent evp;
    mode_t mode = 0;
    /* Initialize the message queue attribute structure */
```

mq_open()

```
mqa.mq_flags = MQ_BLOCK;
mqa.mq_msgsize = 25;
mqa.mq_maxmsg = 4;
/* Create message queues "mqone" and "mqtwo" */
/* Fail if either message queue already exist */
mq1 = mq_open("mqone", O_WRONLY|O_CREAT|O_EXCL, mode, &mqa);
mq2 = mq_open("mqtwo", O_RDONLY|O_CREAT|O_EXCL, mode, &mqa);
if((mq1 < 0) || (mq2 < 0))
    abort();
/* Attach notification request to "mqtwo" */
evp.sigev_signo = SIGUSR1;
mq_notify(mq2, &evp);
/* Create thread attribute and set detached state to undetached */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
/* Create the two threads and execute them */
pthread_create(&t1, &attr, sendmsg, (void *)0);
pthread_create(&t2, &attr, recvmg, (void *)0);
/* Join with thread 1 to ensure that it finishes executing before */
/* continuing */
status = &rv;
pthread_join(t1, (void **)&status);
pthread_join(t2, (void **)&status);
/* Detach threads */
pthread_detach(t1);
pthread_detach(t2);
/* Destroy thread attribute */
pthread_attr_destroy(&attr);
/* Check to see if signal was sent for message queue "mqtwo" */
sigfillset(&sigs);
rv = sigwait(&sigs, &sigval);
/* Close message queues and destroy them */
mq_close(mq1);
mq_unlink("mqone");
mq_close(mq2);
mq_unlink("mqtwo");
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13 §15.2.1
- **mq_close()**, p. 435
- **mq_unlink()**, p. 457

mq_receive()

Synopsis

```
#include <mqueue.h>

ssize_t mq_receive( mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio );
```

Library

rtxLib.a

Description

mq_receive() receives the oldest of the highest priority messages from the message queue named by the descriptor *mqdes*.

If the size of the buffer specified by *msg_len* is less than the **mq_msgsize** attribute of the message queue, **mq_receive()** fails and returns an error. Otherwise, the message named by *mqdes* is copied to the buffer pointed to by *msg_ptr* and removed from the queue.

If the argument *msg_prio* is not NULL, the priority of the selected message will be stored in the field pointed to by *msg_prio*.

If the message queue named by *mqdes* is empty and **O_NONBLOCK** is not set in the message queue description associated with *mqdes*, **mq_receive()** blocks until a message is enqueued on the message queue. If more than one thread is waiting to receive a message when a message arrives at an empty queue, the thread of highest priority that has waited the longest receives the message. If the specified message queue is empty and **O_NONBLOCK** is set in the message queue description associated with *mqdes*, no message is removed from the queue, and **mq_receive()** returns an error with *errno* set to **[EAGAIN]**.

This call is only interrupt handler safe if the message queue is non-blocking.

If successful, **mq_receive()** returns the size of the received message.

If **mq_receive()** fails, no message is removed from the queue, **-1** is returned, and *errno* is set.

Errors

[EAGAIN]	O_NONBLOCK was set in the message description associated with <i>mqdes</i> and the specified message queue is empty.
[EBADF]	<i>mqdes</i> is not a valid message queue descriptor open for reading.
[EMSGSIZE]	The specified message buffer size, msg_len , is less than the message size attribute of the message queue.
[EINTR]	A unblocked signal is pending on the system or the thread.

Example

The following example is a function that receives up to seven messages from a message queue. If the queue is empty, the function does not block on the receive call.

```
#include <mqueue.h>
#include <pthread.h>
#include <fcntl.h>
#include <cLib.h>

void *recvmsg(void *arg)
{
    unsigned int prio;
    int rv = 0;
    char recv_buf[35];
    mq_attr_t mqa;
    mqd_t mq;
    mode_t mode = 0;
    /* Open the message queue "mqone" in read only mode */
    /* Exit the function if the message does not exist */
    mq = mq_open("mqone", O_RDONLY, mode, NULL);
    if(mq < 0)
    {
        rv = -1;
        pthread_exit((void *)&rv);
    }
    /* Check to see if the message queue is non-blocking */
    /* If it is not, change it to non-blocking */
    mq_getattr(mq, &mqa);
    if(mqa.mq_flags == MQ_BLOCK)
    {
        mqa.mq_flags = MQ_NONBLOCK;
        mq_setattr(mq, &mqa, NULL);
    }
    /* Receive up to seven messages from the queue */
    for(rv = 0; rv < 7; rv++)
    {
        memset((void *)recv_buf, '\0', 35);
        if(mq_receive(mq, (char *)recv_buf, (size_t)35, &prio) > 0)
            printf("Message received: %s\n", recv_buf);
        else
        {
            printf("No message received\n");
            break;
        }
    }
}
```

mq_receive()

```
    /* Close message queue */
    mq_close(mq);
    rv = 0;
    pthread_exit((void *)&rv);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13 §15.2.5
- **mq_open()**, p. 441
- **mq_send()**, p. 449

Synopsis

```
#include <mqueue.h>

int mq_send( mqd_t mqdes, const char *msg_ptr, size_t msg_len,
             unsigned int msg_prio );
```

Library

rtxLib.a

Description

mq_send() adds the message pointed to by *msg_ptr* to the message queue specified by *mqdes*.

msg_len specifies the length of the message pointed to by *msg_ptr*. If *msg_len* is greater than the **mq_msgsize** attribute of the message queue, **mq_send()** returns an error.

If the specified message queue is not full, **mq_send()** behaves as if the message is inserted into the message queue at the point indicated by *msg_prio*. A message with larger numeric value of *msg_prio* is inserted before messages with lower values of *msg_prio*. A message is inserted after other messages in the queue, if any, with equal *msg_prio*. The value for *msg_prio* must be in the range 0 through **_POSIX_MQ_PRIO_MAX**.

If the specified message queue is full and **O_NONBLOCK** is not set in the message queue description associated with *mqdes*, **mq_send()** blocks until space becomes available to enqueue the message. If more than one thread is waiting to send when space becomes available in the message queue, the highest priority thread sends its message. If the specified message queue is full and **O_NONBLOCK** is set in the message queue description associated with *mqdes*, the message is not queued, and **mq_send()** returns an error with *errno* set to **[EAGAIN]**.

This call is only interrupt handler safe if the message queue is non-blocking.

If successful, **mq_send()** returns 0. Otherwise, the message is not enqueued, a value of -1 is returned, and *errno* is set to indicate the error.

Errors

[EAGAIN]	O_NONBLOCK is set in the message queue description associated with <i>mqdes</i> , and the specified message queue is full.
[EBADF]	The <i>mqdes</i> argument is not a valid message queue descriptor open for writing.
[EMSGSIZE]	The specified message length, msg_len , exceeds the message size attribute of the message queue.

mq_send()

[EINTR]	A unblocked signal is pending on the system or the thread.
[EINVAL]	The value of <i>msg_prio</i> is outside the valid range.

Example

The following example is a function that sends 6 messages to a message queue and one message to another.

```
#include <mqueue.h>
#include <pthread.h>
#include <stddef.h>
#include <fcntl.h>

void *sendmsg(void *arg)
{
    mqd_t mq;
    int rv = 0;
    mode_t mode = 0;
    /* Open the message queue "mqone" in read/write mode */
    /* Create it if the message queue does not exist yet */
    /* with default attributes */
    mq = mq_open("mqone", O_RDWR | O_CREAT, mode, NULL);
    if(mq < 0)
    {
        rv = -1;
        pthread_exit((void *)&rv);
    }
    /* Write six messages to this message queue */
    mq_send(mq, "Message 1, priority 15", 22, 15);
    mq_send(mq, "Message 2, priority 0", 21, 0);
    mq_send(mq, "Message 3, priority 1", 21, 1);
    mq_send(mq, "Message 4, priority 31", 22, 31);
    mq_send(mq, "Message 5, priority 19", 22, 19);
    mq_send(mq, "Message 6, priority 25", 22, 25);
    /* close this message queue */
    mq_close(mq);
    /* Open the message queue "mqtwo" in read/write mode */
    /* Create it if the message queue does not exist yet */
    /* with default attributes */
    mq = mq_open("mqtwo", O_RDWR | O_CREAT, mode, NULL);
    if(mq < 0)
    {
        rv = -1;
        pthread_exit((void *)&rv);
    }
    /* Write a message to this message queue */
```

```
mq_send(mq, "Notify", 7, 20);  
/* close this message queue */  
mq_close(mq);  
pthread_exit((void *)&rv);  
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	Yes

References

- `mq_open()`, p. 441
- `mq_receive()`, p. 446

mq_setattr()

Synopsis

```
#include <mqueue.h>

int mq_setattr( const mqd_t mqdes,
                const mq_attr_t *mqstat,
                const mq_attr_t *omqstat );
```

Library

rtxLib.a

Description

mq_setattr() sets the attributes associated with the message queue specified by *mqdes*.

The flags value *mq_flags* can be changed at any time, whether or not there are messages on the queue or notification requests attached to the queue. The supported values for the flags field, which are mutually exclusive, follow:

MQ_BLOCK	Blocking message queue.
MQ_NONBLOCK	Non-blocking message queue.

The values of the *mq_maxmsg*, *mq_msgsize*, and *mq_curmsg* members of the **mq_attr** structure are ignored.

If *omqstat* is not NULL, the previous message queue attributes are be stored in this location. These values are the same as if **mq_getattr()** were called.

If successful, **mq_setattr()** returns 0. Otherwise, **mq_setattr()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>mqdes</i> is not a valid message queue descriptor.
[EINVAL]	The value of <i>mq_flags</i> is not valid.

Example

See **mq_receive()**, p. 446.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13 §15.2.7
- **mq_open()**, p. 441
- **mq_send()**, p. 449

Synopsis

```
#include <mqueue.h>

ssize_t mq_timedreceive ( mqd_t mqdes, char *msg_ptr, size_t msg_len,
                        unsigned int *msg_prio,
                        const struct timespec *timeout );
```

Library

rtxLib.a

Description

mq_timedreceive() receives the oldest of the highest priority messages from the message queue named by the descriptor *mqdes*.

If the size of the buffer specified by *msg_len* is less than the **mq_msgsize** attribute of the message queue, **mq_timedreceive()** fails and returns an error. Otherwise, the message from the message queue named by *mqdes* is copied to the buffer pointed to by *msg_ptr* and removed from the queue.

If the argument *msg_prio* is not NULL, the priority of the selected message will be stored in the field pointed to by *msg_prio*.

If the message queue named by *mqdes* is empty and **O_NONBLOCK** is not set in the message queue description associated with *mqdes*, **mq_timedreceive()** blocks until a message is enqueued on the message queue or until the time interval specified in *timeout* has elapsed. If more than one thread is waiting to receive a message when a message arrives at an empty queue, the thread of highest priority that has waited the longest receives the message. If the specified message queue is empty and **O_NONBLOCK** is set in the message queue description associated with *mqdes*, no message is removed from the queue, and **mq_timedreceive()** returns an error with *errno* set to **[EAGAIN]**. If **mq_timedreceive()** blocks and *timeout* elapses before a message is removed from the queue, **mq_timedreceive()** returns an error with *errno* set to **[ETIMEDOUT]**.

This call is only interrupt handler safe if the message queue is non-blocking.

If **mq_timedreceive()** fails, no message is removed from the queue, **-1** is returned, and *errno* is set.

Errors

[EAGAIN]

O_NONBLOCK was set in the message description associated with *mqdes* and the specified message queue is empty.

[EBADF]

mqdes is not a valid message queue descriptor open for reading.

mq_timedreceive()

[EMSGSIZE]	The specified message buffer size, msg_len , is less than the message size attribute of the message queue.
[EINTR]	A unblocked signal is pending on the system or the thread.
[ETIMEDOUT]	Time specified in <i>timeout</i> elapsed before a message was received.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	Yes

References

- **mq_open()**, p. 441
- **mq_timedsend()**, p. 455

Synopsis

```
#include <mqueue.h>

int mq_timedsend( mqd_t mqdes, const char *msg_ptr, size_t msg_len,
                 unsigned int msg_prio, const struct timespec *timeout );
```

Library

rtxLib.a

Description

mq_timedsend() adds the message pointed to by *msg_ptr* to the message queue specified by *mqdes*.

msg_len specifies the length of the message pointed to by *msg_ptr*. If *msg_len* is greater than the **mq_msgsize** attribute of the message queue, **mq_timedsend()** returns an error.

If the specified message queue is not full, **mq_timedsend()** behaves as if the message is inserted into the message queue at the point indicated by *msg_prio*. A message with larger numeric value of *msg_prio* is inserted before messages with lower values of *msg_prio*. A message is inserted after other messages in the queue, if any, with equal *msg_prio*. The value for *msg_prio* must be in the range 0 through **_POSIX_MQ_PRIO_MAX**.

If the specified message queue is full and **O_NONBLOCK** is not set in the message queue description associated with *mqdes*, **mq_timedsend()** blocks until space becomes available to enqueue the message or until the time interval specified in *timeout* has elapsed. If more than one thread is waiting to send when space becomes available in the message queue, the highest priority thread sends its message. If the specified message queue is full and **O_NONBLOCK** is set in the message queue description associated with *mqdes*, the message is not queued, and **mq_timedsend()** returns an error with *errno* set to **[EAGAIN]**. If **mq_timedsend()** blocks and *timeout* elapses before the message is enqueued, the message is not sent and **mq_timedsend()** returns an error with *errno* set to **[ETIMEDOUT]**.

This call is only interrupt handler safe if the message queue is non-blocking.

If successful, **mq_timedsend()** returns 0. Otherwise, the message is not enqueued, a value of -1 is returned, and *errno* is set to indicate the error.

Errors

[EAGAIN]	O_NONBLOCK is set in the message queue description associated with <i>mqdes</i> , and the specified message queue is full.
[EBADF]	The <i>mqdes</i> argument is not a valid message queue descriptor open for writing.

mq_timedsend()

[EMSGSIZE]	The specified message length, msg_len , exceeds the message size attribute of the message queue.
[EINTR]	A unblocked signal is pending on the system or the thread.
[EINVAL]	The value of <i>msg_prio</i> is outside the valid range.
[ETIMEDOUT]	The time set in <i>timeout</i> elapsed without space to send a message.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13 §15.2.4
- **mq_open()**, p. 441
- **mq_timedreceive()**, p. 453

Synopsis

```
#include <mqueue.h>
int mq_unlink( const char *name );
```

Library

rtxLib.a

Description

mq_unlink() removes the message queue named by the path name *name*.

After calling **mq_unlink()** with *name*, a call to **mq_open()** with *name* will fail if the **O_CREAT** flag is not set.

If one or more threads have the message queue open when **mq_unlink()** is called, **mq_unlink()** returns immediately without blocking, but destruction of the message queue is postponed until all references to the message queue have been closed. Calls to **mq_open()** to re-create the message queue will fail until the message queue is actually removed.

If successful, **mq_unlink()** returns 0. Otherwise, **mq_unlink()** returns -1 and sets *errno*. The message queue is unaffected.

Errors

[ENOENT]	The message queue does not exist.
----------	-----------------------------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Example

See **mq_open()**, p. 441.

References

- IEEE Draft Standard 1003.4/D13 §15.2.3
- **mq_close()**, p. 435
- **mq_open()**, p. 441

mtod()

Synopsis

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

mtod( m, t );
```

Library

None. This is a macro defined in `<sys/mbuf.h>`.

Description

mtod() converts the data pointer of the memory buffer (mbuf) given by *m* to a data pointer of the type *t*, which points to the data portion of the same mbuf.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **dtom()**, p. 178

Synopsis

```
#include <sys/mman.h>
int munlock( void *addr, size_t len );
```

Library

devLib.a

Description

munlock() unlocks all pages mapped by OS Open starting at the address pointed to by *addr* for *len*.

Because OS Open does not swap real-memory pages to secondary storage, **munlock()** is not necessary, but is included for compatibility with POSIX application profiles and only checks parameters for validity. The *addr* argument must be on a page boundary.

If successful, **munlock()** returns 0. Otherwise, **munlock()** returns -1 and sets *errno*.

Errors

[EINVAL]	The <i>addr</i> argument does not lie on a page boundary.
----------	---

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13 §12.1.2
- **mlock()**, p. 430

munlockall()

Synopsis

```
#include <sys/mman.h>
int munlockall( void );
```

Library

devLib.a

Description

munlockall() causes all pages mapped by OS Open to be unlocked, that is, not constrained to be in real memory.

Because OS Open does not swap real-memory pages to secondary storage, this function has no meaning, but included for compatibility with POSIX application profiles. It performs no function.

munlockall() returns 0.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13 §12.1.1
- **mlockall()**, p. 431

Synopsis

```
#include <sys/mman.h>
int munmap( void *addr, size_t len );
```

Library

devLib.a

Description

munmap() removes any mappings for pages in the address range starting at the address pointed to by *addr* and continuing for *len* bytes rounded to the next page size.

If there are no mappings in the specified address range, **munmap()** has no effect. The *addr* argument must be on a page boundary.

If successful, **munmap()** returns 0. Otherwise, **munmap()** returns -1 and sets *errno*.

Errors

[EINVAL]	The <i>addr</i> argument is not a multiple of the system page size, or the range given by <i>addr</i> and <i>offset</i> is outside the OS Open address range.
----------	---

Example

See **shm_open()**, p. 734.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13 §12.2.2
- **M_PREPEND()**, p. 396

mutex_dump()

Synopsis

```
#include <kadtLib.h>

int mutex_dump( pthread_mutex_t mutexid, char *buf );
```

Library

kadtLib.a

Description

mutex_dump() displays information about the mutex *mutexid*.

If *buf* is NULL, the output is generated by **bprintf()**. Otherwise, the output is stored in the buffer pointed to by *buf*.

The **mutex_dump()** function returns -1 if the **mutexid** parameter is not a valid mutex, otherwise it returns 0.

Errors

None.

Example

The following shows a sample dump for mutex 0x27f0a8.

```
OS OPEN>mutex_dump(0x27f0a8)
```

```
-----
Mutex ID: 0x27f0a8
Owner: none
Protocol: NO_PRIO_INHERIT
Waiting threads:
-----
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84

Synopsis

```
#include <kadtLib.h>
int mutex_list( char **bufp );
```

Library

kadtLib.a

Description

mutex_list() provides formatted information about all mutexes.

The information is part of the **kda_dump()** display.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

mutex_list() returns 0.

Errors

None.

Example

The following shows a sample listing of mutexes.

```
OS OPEN>mutex_list()
```

Mutex	Owner
0x27f0a8	none
0x4292f8	none
0x4292b0	none
0x27b17c	none
0x277860	none
0x277818	none
0x27af54	none
0x27af0c	none
0x4b88b0	none
0x27ae78	none
0x27ae30	none
0x4b9b98	none
0x4b9b24	none
0x4b9ab0	none
0x27a758	none
0x4bccf4	none
0x4bccac	none
0x4bd7e0	none
0x4bdbf0	none

mutex_list()

```
0x277680  none
0x158574  none
0x158500  none
----- Total Number of Mutexes: 22 -----
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84
- **kda_dump()**, p. 349

Synopsis

```
#include <kadtLib.h>

int mutexattr_dump( pthread_mutexattr_t * attrid, char *buf );
```

Library

kadtLib.a

Description

mutexattr_dump() displays information about the mutex attribute pointed to by *attrid*.

If *buf* is NULL, the output is generated by **bprintf()**. Otherwise, the output is stored in the buffer pointed to by *buf*.

mutexattr_dump() returns -1 if *attrid* does not point to a valid mutex attribute. Otherwise **mutexattr_dump()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84

nanosleep()

Synopsis

```
#include <time.h>

int nanosleep( const struct timespec *rqtp,
               struct timespec *rmtp );
```

Library

rtxLib.a

Description

nanosleep() blocks the calling thread until the interval pointed to by *rqtp* elapses.

The interval may be longer than requested because the *rqtp* argument is rounded up to an integer multiple of the sleep resolution or because OS Open has scheduled other activity. The interval is never shorter than specified by *rqtp*.

If *rmtp* is not NULL, the **timespec** structure that *rmtp* points to is updated to contain the amount of time remaining in the interval if **nanosleep()** is interrupted by a signal.

If successful, **nanosleep()** returns 0. If **nanosleep()** is interrupted by a signal, the function returns -1 and sets *errno* to **[EINTR]**. For all other error conditions, **nanosleep()** returns -1 and sets *errno*.

Errors

[EINTR]	A unblocked signal is pending on OS Open or the thread.
[EINVAL]	The tv_nsec field in the timespec structure pointed to by <i>rqtp</i> contains a value greater than or equal to 1000 million or less than 0.

Example

The following example clears the screen.

```
#include <time.h>
#include <stdio.h>
#include <stddef.h>

#define ESC 033

void clearScreen(void)
{
    struct timespec rqtp;
    /* write escape sequence to clear the screen to stdout */
    putchar(ESC);
    putchar('I');
    putchar('H');
```

```
    putchar(ESC);
    putchar('I');
    putchar('J');
    putchar((char)(10));
    /* pause execution for 50 milliseconds */
    rntp.tv_sec = 0;
    rntp.tv_nsec = 50000000; /* 50 milliseconds */
    nanosleep(&rntp, NULL);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4/D13, §14.2.5

net_init()

Synopsis

```
#include <sys/netLib.h>
int net_init( void );
```

Library

netLib.a

Description

net_init() initializes **netLib.a**.

A **tcip_init()** call must be made before **net_init()** is called. A **net_init()** call must be made before the functions in **netLib.a** can be used. **net_init()** can be called only once during an instance of the OS Open system.

If successful, **net_init()** returns 0. Otherwise, **net_init()** returns -1.

Errors

None.

Example

See **slip_attach()**, p. 756.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **ifconfig()**, p. 304
- **route()**, p. 674
- **tcip_init()**, p. 857

Synopsis

```
#include <pthread.h>
#include <sys/tcpipLib.h>

pthread_mutex *net_splimp();
```

Library

tcpipLib.a

Description

net_splimp() returns a pointer to a locked mutex that is used for serialization within a network protocol stack.

The pointer must be supplied to a corresponding **net_splx()** call.

net_splimp() must be called before calling any function that manipulates network interface data objects or the Internet Protocol (IP) common packet input queue.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **net_splx()**, p. 470

net_splx()

Synopsis

```
#include <pthread.h>
#include <sys/tcpipLib.h>
void net_splx( pthread_mutex_t *s );
```

Library

tcpipLib.a

Description

net_splx() releases the lock on network data structures placed by a previous call to **net_splimp()**.

s points to the value returned by **net_splimp()**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **net_splimp()**, p. 469

Synopsis

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

void nfs_authget(int *uid, int *gid, int *ngids, int **gids);
```

Library

nfsLib.a

Description

nfs_authget() function retrieves UNIX authentication parameters used by Network File System (NFS) library. *uid* specifies user ID, *gid* specifies group ID, *ngids* specifies number of group IDs and *gids* points to an array of group IDs.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **nfs_authset()**, p. 472
- **nfs_dinit()**, p. 473
- **nfs_mount()**, p. 474
- **nfs_showmount()**, p. 475
- **nfs_umount()**, p. 476

nfs_authset()

Synopsis

```
#include <sys/nfsLib.h>
int nfs_authset(int uid, int gid, int ngids, int *gids);
```

Library

nfsLib.a

Description

nfs_authset() sets UNIX authentication parameters used by the Network File System (NFS) library. *uid* specifies the user ID, *gid* specifies the group ID, *ngids* specifies the number of group IDs, and *gids* points to an array of group IDs.

If successful, **nfs_authset()** returns 0. Otherwise, **nfs_authset()** returns -1.

Note: **nfs_authset()** function and **nfs_mount()** function are non-reentrant with respect to each other.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **nfs_authget()**, p. 471
- **nfs_dinit()**, p. 473
- **nfs_mount()**, p. 474
- **nfs_showmount()**, p. 475
- **nfs_umount()**, p. 476

Synopsis

```
#include <sys/nfsLib.h>
int nfs_dinit();
```

Library

nfsLib.a

Description

nfs_dinit() initializes Network File System (NFS) library. **nfs_dinit()**, which can be called only once, must be called before any other function in NFS library.

If successful, **nfs_dinit()** returns 0. Otherwise, **nfs_dinit()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **nfs_authget()**, p. 471
- **nfs_authset()**, p. 472
- **nfs_mount()**, p. 474
- **nfs_showmount()**, p. 475
- **nfs_umount()**, p. 476

nfs_mount()

Synopsis

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

int nfs_mount(char *host, char *remote_filesystem,
              char *local_filesystem, nfsmount_t *opt);
```

Library

nfsLib.a

Description

nfs_mount() mounts *remote_filesystem*, creating a local device *local_filesystem* associated with a remote file system on a specified *host*.

opt specifies additional mount options. Mount options include: read buffer size, from 512 to 8192 bytes; write buffer size, from 512 to 8192 bytes; timeout, in seconds, for each Network File System (NFS) call; retry count; and mount type. If *opt* is NULL, **nfs_mount()** uses default mount options.

Mounts can be of two types: hard (operations retry until successful) or soft (operations retry a specified number of times). **nfsmount_t** structure is described in the **nfsLib.h** include file.

If successful, **nfs_mount()** returns 0. Otherwise, **nfs_mount()** returns -1.

Note: **nfs_authset()** function and **nfs_mount()** function are non-reentrant with respect to each other.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- **nfs_authget()**, p. 471
- **nfs_authset()**, p. 472
- **nfs_dinit()**, p. 473
- **nfs_showmount()**, p. 475
- **nfs_umount()**, p. 476

Synopsis

```
#include <sys/nfsLib.h>
int nfs_showmount(char *args);
```

Library

nfsLib.a

Description

nfs_showmount() displays exported filesystems on a remote host.

args points to a character string specifying options for the **nfs_showmount()** function. *nfs_showmount()* accepts following arguments:

host	Prints a list of exported directories.
-a host	Prints remote mount in format host:directory.
-d host	Lists only directories remotely mounted by clients.
-e host	Prints a list of exported directories.

If successful, **nfs_showmount()** returns 0. Otherwise, **nfs_showmount()** returns -1.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- **nfs_authget()**, p. 471
- **nfs_authset()**, p. 472
- **nfs_dinit()**, p. 473
- **nfs_mount()**, p. 474
- **nfs_umount()**, p. 476

nfs_umount()

Synopsis

```
#include <sys/nfsLib.h>
int nfs_umount(char *local_filesys);
```

Library

nfsLib.a

Description

nfs_umount() unmounts *local_filesys* if no files in the file system are open.

If successful, **nfs_umount()** returns 0. Otherwise, **nfs_umount()** returns -1.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- **nfs_authget()**, p. 471
- **nfs_authset()**, p. 472
- **nfs_dinit()**, p. 473
- **nfs_mount()**, p. 474
- **nfs_showmount()**, p. 475

Synopsis

```
#include <pthread.h>
int np_compare_swap( int *value, int compare, int swap );
```

Library

rtxLib.a

Description

np_compare_swap() compares the data pointed to by *value* with *compare*. If the two are equal, the data pointed to by *value* is replaced by *swap*. The comparison and exchange are atomic with respect to external interrupts and context switches.

If the data is swapped, **np_compare_swap()** returns 1. Otherwise, **np_compare_swap()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

open()

Synopsis

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

int open( const char *path, int oflag, ... );
```

Library

devLib.a

Description

open() establishes the connection between the file named in *path* and a file descriptor.

open() returns the numerically lowest file descriptor that is not open in the device subsystem of OS Open. The *path* parameter must consist of fewer than **PATH_MAX** characters from the POSIX portable file name set. The file status flags and access modes are set according to the *oflag* parameter. **PATH_MAX** is defined in the file **<limits.h>**.

Note: Device subsystem support for **open()** performs a basic parameter consistency check. The primary function of the support is to locate the correct device driver instance and call the device specific **open()**. Individual devices may vary in their interpretation and level of scrutiny of any particular flag. Flag constants are defined in the file **<fcntl.h>**. Errors described in this entry reflect the conditions detected by the device subsystem support.

The three file access modes in *oflag* are mutually exclusive; only one should be specified.

[O_RDONLY]	Open the file for reading only
[O_WRONLY]	Open the file for writing only
[O_RDWR]	Open the file for reading and writing.

Any combination of the remaining flags may be specified; however, some may have no meaning for a given device driver.

[O_APPEND]	Set the file offset to the end of the file before each write
[O_CREAT]	Requires specification of a third parameter, <i>mode</i> , of type mode_t . If the file exists, this flag has no effect. The <i>mode</i> value specifies the file permission bits.
[O_EXCL]	If this flag and O_CREAT are set, open() will fail if the file already exists.
[O_NOCTTY]	If set and <i>path</i> identifies a terminal device, open() does not cause the terminal to be the controlling terminal for OS Open.

[O_NONBLOCK] If the device supports non-blocking opens, this flag causes **open()** to return immediately. Otherwise, **open()** blocks until the device is ready or available.

[O_TRUNC] For regular files successfully opened using **O_RDWR** or **O_WRONLY**, this flag truncates the file to zero length.

If successful, **open()** returns a nonnegative integer file descriptor. Otherwise, **open()** returns `-1` and sets *errno*.

Errors

[EEXIST] **O_CREAT** and **O_EXCL** were specified for a device file.

[ENFILE] There are no more free file descriptors in the system. The number of open files exceeds the number specified when the device subsystem was initialized.

[ENAMETOOLONG] The length of the *path* string exceeds **PATH_MAX**.

[ENOENT] The named file does not exist.

[EINVAL] More than one of **O_WRONLY**, **O_RDWR**, and **O_RDONLY** was specified, or the string pointed to by *path* contains characters not in the portable file name set.

Example

See **read()**, p. 633.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.3.1
- **close()**, p. 127
- **dev_io_init()**, p. 171

opendir()

Synopsis

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

DIR *opendir( const char *dirname );
```

Library

devLib.a

Description

opendir() opens a directory stream corresponding to the directory name pointed to by *dirname*.

The directory is positioned at the first entry.

If successful, **opendir()** returns a pointer to the directory stream. Otherwise, a value of NULL is returned and *errno* is set.

Errors

[EACCES]	Search permission denied for a component of the path, or read permission denied for the directory itself.
[EIO]	A physical input/output (I/O) error occurred.
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of <i>dirname</i> exceeds {PATH_MAX} .
[ENOENT]	<i>dirname</i> does not exist.
[ENOTDIR]	A component of <i>dirname</i> is not a directory.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.1.2.2

Synopsis

```
#include <ioLib.h>
void outbyte( unsigned char *port, unsigned char data );
```

Library

ioLib.a

Description

outbyte() writes the byte containing *data* to the I/O port specified by *port*.
After the byte is written, the PowerPC **eieio** instruction is issued to enforce in-order execution of I/O.
outbyte() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inbyte()**, p. 307
- **inshort()**, p. 319
- **inword()**, p. 331
- **outshort()**, p. 482
- **outword()**, p. 483

outshort()

Synopsis

```
#include <ioLib.h>

void outshort( unsigned short *port, unsigned short data );
```

Library

ioLib.a

Description

outshort() writes the halfword containing *data* to the I/O port specified by *port*.

After the halfword is written, the PowerPC **eieio** instruction is issued to enforce in-order execution of I/O.

outshort() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inbyte()**, p. 307
- **inshort()**, p. 319
- **inword()**, p. 331
- **outbyte()**, p. 481
- **outword()**, p. 483

Synopsis

```
#include <ioLib.h>

void outword( unsigned long *port, unsigned long data );
```

Library

ioLib.a

Description

outword() writes a word containing *data* to the effective address specified by *port*.

After the word is written, the PowerPC **ei** instruction is issued to enforce in-order execution of I/O.

outword() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **inbyte()**, p. 307
- **inshort()**, p. 319
- **inword()**, p. 331
- **outbyte()**, p. 481
- **outshort()**, p. 482

package_install()

Synopsis

```
#include <sys/ksyscall.h>
int package_install( const char *name, void *anchor, void **old_anchor );
```

Library

devLib.a

Description

package_install() provides a standard way to associate a name with a data pointer.

OS Open problem determination software uses **package_install()** to locate data areas in dynamically installed subsystems.

name points to a string that identifies the package. The value pointed to by *anchor* is any value to be associated with the package, such as a global data structure or dynamically allocated memory. If *old_anchor* is not NULL, any anchor value associated with *name* is placed in the location pointed to by *old_anchor*. If *name* did not previously exist, the value of *old_anchor* is NULL.

If successful, **package_install()** returns 0. Otherwise, **package_install()** returns -1 and sets *errno*.

Errors

[ENOMEM]	Internal control blocks could not be allocated.
----------	---

Example

See **sem_open()**, p. 700.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **library_list()**, p. 382

Synopsis

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

int driver_install( int *dev_handle, pata_init );
```

Library

pataLib.a

Description

pata_init() initializes PCMCIA ATA/IDE device driver support, in **pataLib.a**, for 105MB and 260MB hard disk drives manufactured by IBM and Integral.

driver_install() installs **pataLib.a**. In the call to **driver_install()**, *dev_handle* is a pointer to the device handle; *pata_init* is the address of **pata_init()**, which contains the device driver initialization code.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **driver_install()**, p. 177
- Chapter 11, "OS Open PCMCIA Support," in *OS Open User's Guide*

pathconf()

Synopsis

```
#include <unistd.h>

long pathconf( const char *path, int name );
```

Library

devLib.a

Description

pathconf() retrieves the value of a configurable limit or option specified by *name* and associated with the file pointed to by *path*.

The configurable POSIX path name variables are:

_PC_LINK_MAX	Number of links per directory entry
_PC_MAX_CANON	Number of bytes in a terminal canonical input line (terminal special files only)
_PC_MAX_INPUT	Minimum number of bytes available in a terminal input queue (terminal special files only)
_PC_NAME_MAX	Maximum number of bytes in a file name
_PC_PATH_MAX	Maximum number of bytes in a path name
_PC_CHOWN_RESTRICTED	chown() is restricted to a thread with appropriate privileges
_PC_NO_TRUNC	Path name components longer than {NAME_MAX} generate an error
_PC_VDISABLE	Disable terminal special characters

If no limit is associated with *name*, **pathconf()** returns `-1` without altering *errno*. Otherwise, **pathconf()** returns `-1` and sets *errno*.

Errors

[EACCES]	Search permission denied for a component of the path.
[EINVAL]	The value of <i>name</i> is not valid.
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of <i>path</i> exceeds {PATH_MAX} .
[ENOENT]	<i>path</i> names a file that does not exist or points to an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.

Example

See **link()**, p. 384.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.7.1
- **fpathconf()**, p. 217

perror()

Synopsis

```
#include <stdio.h>
void perror( const char *string );
```

Library

fsLib.a

Description

perror() prints an error message to the standard error stream.

If *string* is not NULL and does not point to a NULL character, the string pointed to by *string* is printed to the standard error stream, followed by a colon and a space. Then the message associated with the value in *errno* is printed, followed by a newline.

To produce accurate results, **perror()** should be called immediately after a library routine returns with an error. Otherwise, subsequent calls may alter *errno*.

perror() returns nothing.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **puts()**, p. 615.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.10.4
- **clearerr()**, p. 99
- **ferror()**, p. 196
- **fs_init()**, p. 231
- **strerror()**, p. 794

Synopsis

```
#include <sys/netLib.h>
int ping( char *cmd );
```

Library

netLib.a

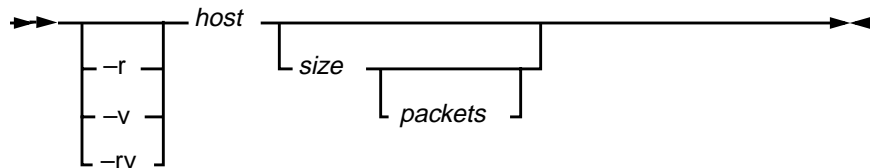
Description

ping() sends an Internet Control Message Protocol (ICMP) **ECHO_REQUEST** to a network host to obtain an ICMP **ECHO_RESPONSE** from a host or gateway.

ping() is useful for:

- Determining network and various foreign host status
- Tracking and isolating hardware and software problems
- Testing, measuring, and managing networks

cmd points to a string of the following format:



If *host* is working and on the network, *host* responds to the echo request. An echo request contains an Internet Protocol (IP) and ICMP header, followed by a **timeval** structure and enough bytes to fill the packet.

ping() sends one datagram per second and prints one line of output for every received response. **ping()** calculates round-trip times and packet loss statistics and displays a brief summary. **ping()** finishes when the calling program times out, the number of packets specified by *packets* is received, or on receipt of a **SIGINT** signal.

The **-r** parameter bypasses the routing tables and sends directly to a *host* on an attached network. If *host* is not on a directly connected network, **ping()** returns an error message. This option can ping a local host through an interface that no longer has a route through it.

The **-v** parameter displays verbose information about the received packet. Verbose information includes 20 bytes for the IP header of the packet, followed by 8 bytes for the ICMP header, followed by the ICMP message data. The first 8 bytes of the message data represent a time value. The remaining data begins with byte value X'08'; each successive byte value is incremented by 1. Up to 56 bytes of message data for each packet are displayed.

ping()

The **-rv** parameter combines the effects of **-r** and **-v**.

size specifies the size of the packet to be sent. By default, *size* is 64 bytes, of which 8 bytes is ICMP header data. The maximum *size* is 4096 bytes.

If successful, **ping()** returns 0. Otherwise, **ping()** returns -1.

Errors

None.

Example

See **slip_attach()**, p. 756.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- **gethostbyname()**, p. 265
- **getnetbyname()**, p. 272
- **ifconfig()**, p. 304
- **net_init()**, p. 468
- **route()**, p. 674

Synopsis

```
#include <netinet/in.h>
#include <rpc/rpc.h>

struct pmaplist *pmap_getmaps( struct sockaddr_in *addr );
```

Library

rpcLib.a

Description

pmap_getmaps() uses the portmap daemon to return a list of current RPC program-to-port mappings on the host located at the IP address pointed to by *addr*.

If no list exists or the portmap daemon could not be contacted, **pmap_getmaps()** returns a NULL.

Note: The **rpcinfo -p** command calls the subroutine.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **pmap_set()**, p. 495
- **pmap_unset()**, p. 496
- **rpc_thread_init()**, p. 676
- **svc_register()**, p. 824

pmap_getport()

Synopsis

```
#include <types.h>
#include <netinet/in.h>
#include <rpc/rpc.h>
u_short pmap_getport( struct sockaddr_in *addr, u_long prognum,
                      u_long versnum, u_long protocol );
```

Library

rpcLib.a

Description

pmap_getport() causes the portmap daemon to return the port number on which a service waits.

addr points to the IP address of the host supporting the remote program providing the service.

prognum specifies the program number of the remote procedure; *versnum* specifies its version number.

protocol, which specifies the transport protocol used by the service, must have a value of **IPPROTO_TCP** or **IPPROTO_UDP**. These constants are defined in the file **<netinet.h/in.h>**.

If the mapping does not exist or the remote portmap daemon could not be contacted, **pmap_getport()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <types.h>
#include <netinet/in.h>
#include <rpc/rpc.h>

enum clnt_stat pmap_rmtcall( struct sockaddr_in *addr, u_long prognum,
                           u_long versnum, u_long procnum,
                           xdrproc_t inproc, char * in,
                           xdrproc_t outproc, char * out,
                           struct timeval tout, u_long *portp );
```

Library

rpcLib.a

Description

pmap_rmtcall() causes the portmap daemon to make a remote procedure call.

Clients use the portmap daemon to determine the ports where remote procedure calls are sent. *addr* points to the IP address of the host on which the remote program supporting the waiting service is accessed.

prognum specifies the program number of the remote procedure; *versnum* specifies its version number.

procnum specifies the procedure to be called.

inproc specifies the XDR routine that encodes RPC procedure parameters.

outproc specifies the XDR routine that decodes the RPC procedure results.

in points to the address of the procedure arguments.

out points to the address where the results are placed.

tout sets the time the routine waits for results before resending the call.

portp points to the program port number if the procedure succeeds.

pmap_rmtcall() returns a value of **enum clnt_stat**, which **clnt_perrno()** can translate into a displayed message.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

pmap_rmtcall()

References

- `clnt_broadcast()`, p. 101
- `rpc_thread_init()`, p. 676

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t pmap_set( u_long prognum, u_long versnum, u_long protocol,
                u_short port );
```

Library

rpcLib.a

Description

pmap_set() causes the portmap daemon to map the program number, version number, and protocol of a remote procedure call to a port on the machine's portmap daemon.

Note: **pmap_set()** is called by **svc_register()**.

prognum specifies the program number of the remote program; *versnum* specifies its version number.

protocol specifies the transport protocol used by the service. The values for this parameter must be **IPPROTO_UDP** or **IPPROTO_TCP**. These constants are defined in the file **<netinet.h/in.h>**.

port specifies the port on the machine's portmap daemon.

If successful, **pmap_set()** returns 1. Otherwise, **pmap_set()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **pmap_getmaps()**, p. 491
- **pmap_unset()**, p. 496
- **rpc_thread_init()**, p. 676
- **svc_register()**, p. 824

pmap_unset()

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t pmap_unset( u_long prognum, u_long versnum );
```

Library

rpcLib.a

Description

pmap_unset() destroys mappings between the program number and version number of a remote procedure call and the ports on the host portmap daemon.

prognum specifies the program number of the remote procedure; *versnum* specifies its version number.

If successful, **pmap_unset()** returns 1. Otherwise, **pmap_unset()** returns 0.

References

- **pmap_getmaps()**, p. 491
- **pmap_set()**, p. 495
- **rpc_thread_init()**, p. 676
- **svc_register()**, p. 824

Synopsis

```
#include <kadtLib.h>
int pool_list( char **bufp );
```

Library

kadtLib.a

Description

pool_list() provides formatted information about all memory pools.

The information is part of the **kda_dump()** display.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

pool_list() returns 0.

Errors

None.

Example

The following shows a sample listing of memory pools.

```
OS OPEN>pool_list()
Memory pool   Number    Size   High water mark
4694676       1000      128        8
----- Total Number of Memory Pools: 1 -----
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84
- **kda_dump()**, p. 349

portmap_thread()

Synopsis

```
#include <rpc/rpcLib.h>
int portmap_thread(void);
```

Library

rpcLib.a

Description

portmap_thread() starts the portmap daemon.

The portmap daemon is required if a machine is running any RPC servers. Only one instance of the portmap daemon can run on a system.

If successful, **portmap_thread()** returns 0. Otherwise, **portmap_thread()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getrpcport()**, p. 288
- **pmap_getport()**, p. 492
- **rpc_thread_init()**, p. 676

Synopsis

```
#include <math.h>

double pow( double x, double y );
```

Library

mathLib.a

Description

pow() returns x raised to the power y .

If x is negative and y is not an integral value, or if x is 0 and y is less than or equal to 0, a domain error occurs and **pow()** sets *errno*. If the result is out of range, a range error occurs and **pow()** sets *errno*.

Errors

[EDOM]	x is negative and y is not an integral value, or, x is 0 and y is less than or equal to 0.
[ERANGE]	The result is out of range.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.5.1

printf()

Synopsis

```
#include <stdio.h>
int printf( const char *format-string, ... );
```

Library

fsLib.a

Description

printf() formats and prints a series of characters and values to the standard output.

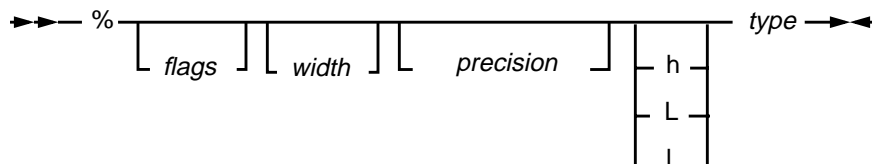
format-string points to an object comprising ordinary characters, escape sequences, and format specifications. The ordinary characters are copied in order of their appearance to the standard output. Format specifications, beginning with a percent sign (%), determine the output format for any argument list following *format-string*.

Notes: The file system must have been initialized by **fs_init()**.

The results are indeterminate if an attempt is made to print more than 509 characters.

format-string is read from left to right. When the first format specification is found, the value of the first argument after *format-string* is converted and output according to the format specification. A second format specification converts and outputs the second argument after *format-string*, and so on through the end of the *format-string*. If there are more arguments than format specifications, the extra arguments are evaluated and ignored. The results are undefined if there are not enough arguments for all the format specifications.

A format specification has the following form:



Each field in the format specification is a single character or number signifying a particular format option. The **type** character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, a number, or pointer. The simplest format specification contains only the percent sign and a **type** character (for example, "%s").

Note: All options in the tables marked with an at sign (@) are not supported in the OS Open system. **printf()** interprets these options as literal characters and prints them as they appear in *format-string*.

The following optional fields control other aspects of the formatting:

<i>flags</i>	Justification of output; printing of signs, blanks, decimal points, and octal and hexadecimal prefixes; semantics for the wchar_t precision unit
<i>width</i>	Minimum number of characters (bytes) output
<i>precision</i>	Maximum number of characters (bytes) printed for all or part of the output field, or minimum number of digits printed for integer values.
h	A prefix with d , i , o , u , x , X , and n types that specifies that the argument is a short int or unsigned short int .
l	A prefix with d , i , o , u , x , X , and n types that specifies that the argument is a long int or unsigned long int .
L@	A prefix with e , E , f , g , or G types that specifies that the argument is a long double .

If a percent sign (%) is followed by a character that has no meaning as a format field, the character is simply copied out to the standard output. To print a percent sign, use "%%".

The **type** characters and their meanings are given in the following table:

Table 41. printf() Format Types

Character	Argument	Output Format
d, i	Integer	Signed decimal integer
u	Integer	Unsigned decimal integer
o	Integer	Unsigned octal integer
x	Integer	Unsigned hexadecimal integer, using "abcdef"
X	Integer	Unsigned hexadecimal integer, using "ABCDEF"
f	Floating-point	Signed value of the form <code>[-]dddd.dddd</code> , where <i>ddd</i> is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number. The number of digits after the decimal point equals the requested precision.
e	Floating-point	Signed value of the form <code>[-]d.ddd e[sign] ddd</code> , where <i>d</i> is a single-decimal digit, <i>ddd</i> is one or more decimal digits, <i>ddd</i> is exactly three decimal digits, and <i>sign</i> is + or -.

Table 41. printf() Format Types

Character	Argument	Output Format
E	Floating-point	Identical to the e format, except that E introduces the exponent instead of e .
g	Floating-point	Signed value of the f or e format, whichever is more compact for the given value and <i>precision</i> . The e format is used only when the exponent of the value is less than -4 or greater than <i>precision</i> . Trailing zeros are truncated and the decimal point appears only if one or more digits follow it.
G	Floating-point	Identical to the g format, except that E introduces the exponent instead of e .
c	Character	Single character.
s	String	Characters printed up to the first NULL (<code>\0</code>) or until <i>precision</i> is reached.
n	Pointer to integer	Number of characters successfully written so far to stream or buffer; the value is stored in the integer whose address is given as the argument.
p	Pointer	Pointer to void converted to a sequence of printable characters. The value will have the format of a hexadecimal number beginning with "0x" and a precision of eight, padded with zeros.
lc@	Wide character	Multibyte character.
ls@	Wide string	Multibyte characters printed up to the first wchar_t NULL (<code>L'\0'</code>) or until <i>precision</i> is reached.

flag characters and their meanings follow. Note that more than one *flag* can appear in a format specification:

Table 42. Optional printf() Flags

Flag	Meaning	Default
–	Left-justify the result within the field width.	Right-justify
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative signed values

Table 42. Optional printf() Flags

Flag	Meaning	Default
space(' ')	Prefix the signed positive output value with a blank. The + flag overrides the <i>blank</i> flag if both appear, and a positive signed value is output with a sign.	No blank
#	When used with the 0 , x , or X formats, the # flag prefixes any nonzero output value with 0 , 0x , or 0X respectively.	No prefix
	When used with the f , e , or E formats, the # flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
	When used with the g or G formats, the # flag forces the output value to contain a decimal point and prevents truncation of trailing zeros.	Decimal point appears only if digits follow it; trailing zeros are truncated.
	When used with the ls format, the # flag causes <i>precision</i> to be measured in wchar_t characters.	<i>precision</i> indicates the maximum number of bytes to output.
0 (zero)	When used with the d , i , o , u , x , X , e , E , f , g , or G formats, the 0 flag causes leading 0's to pad the output to the field width. The 0 flag is ignored if precision is specified for an integer or if the - flag is specified.	Space padding.

The # flag should not be used with **c**, **lc**, **d**, **i**, **u**, **s**, or **p** types.

width is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of output characters is smaller than *width*, blanks or zeroes are added on the left or right (depending on whether the **-** flag is specified) until *width* is reached.

width never causes a value to be truncated. If the number of characters in the output value is greater than *width*, or *width* is not given, all characters in the output value are printed, up to *precision*.

For **ls** type, *width* is specified in bytes. If the number of bytes in the output value is less than *width*, single byte blanks are added on the left or right, depending on whether the **-** flag is specified, until the minimum width is reached.

width can be an asterisk (*), in which case an argument from *argument-list* supplies the value. *width* argument must precede the value being formatted in *argument-list*.

precision is a non-negative decimal integer preceded by a period (.) that specifies the number of characters to be printed or the number of decimal places. Unlike *width*, *precision* can cause truncation of the output value or rounding of a floating-point value.

precision may be an asterisk(*), in which case an argument from *argument-list* supplies the value. *precision* argument must precede the value being formatted in the argument list.

Interpretation of *precision* and the default when *precision* is omitted depend upon the **type**, as shown in the following table:.

Table 43. Precision Based on printf() Types

Type	Meaning	Default
i d u o x X	<i>precision</i> specifies the minimum number of digits to be printed. If the number of digits in the argument is less than <i>precision</i> , the output value is padded on the left with zeros. The value is truncated when the number of digits exceeds <i>precision</i> .	If <i>precision</i> is 0 or omitted, or if a period (.) appears without a number following it, <i>precision</i> is set to 0.
f e E	<i>precision</i> specifies the number of digits to be printed after the decimal point. The last digit printed is rounded.	<i>precision</i> is six. If <i>precision</i> is zero or a period (.) appears without a number following it, no decimal point is printed.
g G	<i>precision</i> specifies the maximum number of significant digits printed.	All significant digits are printed.
c	No Effect	Character printed.
lc	No Effect	wchar_t character printed.
s	<i>precision</i> specifies the maximum number of characters to be printed. Characters in excess of <i>precision</i> are not printed.	Characters are printed until a NULL is encountered.

Table 43. Precision Based on printf() Types

Type	Meaning	Default
ls	<i>precision</i> specifies the maximum number of bytes to be printed. Bytes in excess of <i>precision</i> are not printed, however DBCS integrity is always preserved. Output is padded with single byte spaces to the specified precision if less than <i>precision</i> bytes were written out.	wchar_t characters are printed until a NULL character is encountered.

Errors

None.

Example

See **fopen()**, p. 210.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.6.3

pthread_attr_destroy()

Synopsis

```
#include <pthread.h>
int pthread_attr_destroy( pthread_attr_t *attr );
```

Library

rtxLib.a

Description

pthread_attr_destroy() destroys the thread attributes object pointed to by *attr*.

Subsequent use of the destroyed thread attributes object produces errors. The thread attributes object must be reinitialized using **pthread_attr_init()** before being passed as an argument to any pthread function calls.

If successful, **pthread_attr_destroy()** returns 0. Otherwise, **pthread_attr_destroy()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §16.1.1
- **pthread_attr_init()**, p. 519

Synopsis

```
#include <pthread.h>

int pthread_attr_getallocstack_np( const pthread_attr_t *attr,
                                   unsigned long *stackalloc);
```

Library

rtxLib.a

Description

pthread_attr_getallocstack_np() gets the value of the **stackalloc** attribute from the thread attributes object pointed to by *attr*. The value is either **PTHREAD_ALLOCSTACK_NP**, which specifies that the new thread will be created with a pre-allocated stack, or **PTHREAD_EXTENDSTACK_NP**, which specifies that the new thread will be created with a partially allocated stack.

The thread attributes object must be initialized by **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_getallocstack_np()** returns 0. Otherwise, **pthread_attr_getallocstack_np()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **pthread_attr_init()**, p. 519
- **pthread_attr_destroy()**, p. 506
- **pthread_attr_setallocstack_np()**, p. 520

pthread_attr_getdetachstate()

Synopsis

```
#include <pthread.h>

int pthread_attr_getdetachstate( const pthread_attr_t *attr,
                                int *detachstate );
```

Library

rtxLib.a

Description

pthread_attr_getdetachstate() gets the value of the **detachstate** attribute from the thread attributes object pointed to by *attr*.

The thread attributes object must be initialized using **pthread_attr_init()**, or unpredictable results may occur.

If successful, **pthread_attr_getdetachstate()** returns 0 and stores one of the following values in the field pointed to by *detachstate*:

- PTHREAD_CREATE_JOINABLE
- PTHREAD_CREATE_DETACHED

These constants are defined in **<pthread.h>**.

If an error occurs, **pthread_attr_getdetachstate()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §16.1.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

Synopsis

```
#include <pthread.h>

int pthread_attr_getfp_np( pthread_attr_t *attr, int *fpflag );
```

Library

rtxLib.a

Description

pthread_attr_getfp_np() stores the floating point availability flag of the thread attributes object pointed to by *attr* into the location pointed to by *fpflag*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_getfp_np()** returns 0. Otherwise, **pthread_attr_getfp_np()** returns an error number.

Errors

EINVAL	The object pointed to by <i>attr</i> does not exist.
--------	--

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **pthread_attr_init()**, p. 519
- **pthread_attr_setfp_np()**, p. 522
- **pthread_create()**, p. 552

pthread_attr_getinheritsched()

Synopsis

```
#include <pthread.h>

int pthread_attr_getinheritsched( const pthread_attr_t *attr,
                                  int *inheritsched );
```

Library

rtxLib.a

Description

pthread_attr_getinheritsched() gets the value of the **inheritsched** attribute from the thread attributes object pointed to by *attr*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_getinheritsched()** returns 0 and stores one of the following values in the field pointed to by *inheritsched*:

- PTHREAD_EXPLICIT_SCHED
- PTHREAD_INHERIT_SCHED

These constants are defined in the file **<pthread.h>**.

If an error occurs, **pthread_attr_getinheritsched()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.3.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

Synopsis

```
#include <pthread.h>

void pthread_attr_getpr_np( unsigned long *privilege);
```

Library

rtxLib.a

Description

pthread_attr_getpr_np() gets the privilege status of the current thread and stores it in the variable pointed to by *privilege*.

Privilege status can be **PTHREAD_PRIVILEGED_NP** or **PTHREAD_APPLICATION_NP**.

During a system call, thread privilege is set to **PTHREAD_PRIVILEGED_NP**. On exit from the system call, thread privilege is restored to its original value.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

References

- **pthread_attr_gettg_np()**, p. 517
- **pthread_attr_init()**, p. 519
- **pthread_attr_settg_np()**, p. 530
- **pthread_create()**, p. 552
- **pthread_tgcreate_np()**, p. 598

pthread_attr_getschedparam()

Synopsis

```
#include <pthread.h>

int pthread_attr_getschedparam( const pthread_attr_t *attr,
                                struct sched_param *param );
```

Library

rtxLib.a

Description

pthread_attr_getschedparam() gets the scheduling parameter attributes from the thread attributes object pointed to by *attr* and stores them in the structure pointed to by *param*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_getschedparam()** returns 0. Otherwise, **pthread_attr_getschedparam()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Example

See **pthread_attr_setschedparam()**, p. 524.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.3.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

Synopsis

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_getschedpolicy( const pthread_attr_t *attr,
                                int *policy );
```

Library

rtxLib.a

Description

pthread_attr_getschedpolicy() gets the value of the **schedpolicy** attribute from the thread attributes object pointed to by *attr*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_getschedpolicy()** returns 0 and stores one of the following values in the field pointed to by *policy*:

- SCHED_FIFO
- SCHED_RR
- SCHED_OTHER

SCHED_FIFO, SCHED_RR and SCHED_OTHER can be modified by ORing with SCHED_NOPREEMPT flag. These constants are defined in the file **<sched.h>**.

If an error occurs, **pthread_attr_getschedpolicy()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.3.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

pthread_attr_getscope()

Synopsis

```
#include <pthread.h>

int pthread_attr_getscope( pthread_attr_t *attr,
                          int *scope );
```

Library

rtxLib.a

Description

pthread_attr_getscope() gets the value of the **contentionscope** attribute from the thread attributes object pointed to by *attr*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_getscope()** returns 0 and stores one of the following values in the field pointed to by *scope*:

- PTHREAD_SCOPE_SYSTEM
- PTHREAD_SCOPE_PROCESS

These constants are defined in the file **<pthread.h>**.

If an error occurs, **pthread_attr_getscope()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Example

See **pthread_attr_setschedparam()**, p. 524.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.3.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

Synopsis

```
#include <pthread.h>

int pthread_attr_getstackaddr( const pthread_attr_t *attr,
                              void **stackaddr );
```

Library

rtxLib.a

Description

pthread_attr_getstackaddr() gets the value of the **stackaddr** attribute from the thread attributes object pointed to by *attr*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_getstackaddr()** returns 0. Otherwise, **pthread_attr_getstackaddr()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Example

See **pthread_attr_setschedparam()**, p. 524.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §16.1.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

pthread_attr_getstacksize()

Synopsis

```
#include <pthread.h>

int pthread_attr_getstacksize( const pthread_attr_t *attr,
                              size_t *stacksize );
```

Library

rtxLib.a

Description

pthread_attr_getstacksize() gets the value of the **stacksize** attribute, measured in bytes, from the thread attributes object pointed to by *attr*, and stores the value of the **stacksize** attribute in the field pointed to by *stacksize*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_getstacksize()** returns 0. Otherwise, **pthread_attr_getstacksize()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Example

See **pthread_attr_setschedparam()**, p. 524.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §16.1.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

Synopsis

```
#include <pthread.h>

int pthread_attr_gettg_np( const pthread_attr_t *attr,
                           unsigned long *tg);
```

Library

rtxLib.a

Description

pthread_attr_gettg_np() gets the value of the thread group attribute from the thread attributes object pointed to by *attr* and stores it in the variable pointed to by *tg*.

The returned value can be one of the following: **PTHREAD_CURRENT_TG_NP**, **PTHREAD_NEW_TG_NP**, or **PTHREAD_SET_TG_NP**.

The thread attributes object must be initialized by **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_gettg_np()** returns 0. Otherwise, **pthread_attr_gettg_np()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **pthread_attr_init()**, p. 519
- **pthread_attr_settg_np()**, p. 530
- **pthread_attr_gettgid_np()**, p. 518
- **pthread_attr_settgid_np()**, p. 532
- **pthread_create()**, p. 552

pthread_attr_gettgid_np()

Synopsis

```
#include <pthread.h>

int pthread_attr_gettgid_np( const pthread_attr_t *attr,
                             tg_t *tgid);
```

Library

rtxLib.a

Description

pthread_attr_gettgid_np() gets the thread group ID attribute from the thread attributes object pointed to by *attr* and stores it in the variable pointed to by *tgid*.

pthread_attr_settgid_np() must be performed before **pthread_attr_gettgid_np()** for the value returned in *tgid* to be valid.

The thread attributes object must be initialized by **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_gettgid_np()** returns 0. Otherwise, **pthread_attr_gettgid_np()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **pthread_attr_init()**, p. 519
- **pthread_attr_gettg_np()**, p. 517
- **pthread_attr_settg_np()**, p. 530
- **pthread_attr_settgid_np()**, p. 532
- **pthread_create()**, p. 552
- **pthread_tgcreate_np()**, p. 598

Synopsis

```
#include <pthread.h>

int pthread_attr_init( pthread_attr_t *attr );
```

Library

rtxLib.a

Description

pthread_attr_init() creates and initializes a thread attributes object and stores an identifier for the newly-created object in the field pointed to by *attr*.

The attributes in the newly-created threads attributes object are set to system default values, which are defined in the file **<pthread.h>**.

pthread_attr_init() always returns 0.

Errors

None.

Example

See **mq_open()**, p. 441.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §16.1.1
- **pthread_attr_destroy()**, p. 506
- **pthread_create()**, p. 552

pthread_attr_setallocstack_np()

Synopsis

```
#include <pthread.h>

int pthread_attr_setallocstack_np( pthread_attr_t *attr,
                                   unsigned long stackalloc);
```

Library

rtxLib.a

Description

pthread_attr_setallocstack_np() sets the stack allocation flag to the value of the *stackalloc* parameter in the thread attributes object pointed to by *attr*.

The value of *stackalloc* must be either **PTHREAD_ALLOCSTACK_NP**, which specifies that the new thread will be created with a pre-allocated stack, or **PTHREAD_EXTENDSTACK_NP**, which specifies that the new thread will be created with a partially allocated stack.

If the entire stack is not allocated, the thread is initially allocated one real page for its stack. If the thread needs additional stack space during its execution, additional real pages are allocated, up to the maximum specified in the stack size thread attribute. If this operation exhausts system free memory, the thread is suspended until additional free system memory becomes available.

The thread attributes object must be initialized by **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_setallocstack_np()** returns 0. Otherwise, **pthread_attr_setallocstack_np()** returns an error number.

Errors

[EINVAL]	The value of <i>stackalloc</i> is invalid or the object pointed to by <i>attr</i> does not exist.
----------	---

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **pthread_attr_destroy()**, p. 506
- **pthread_attr_getallocstack_np()**, p. 507
- **pthread_attr_init()**, p. 519

Synopsis

```
#include <pthread.h>

int pthread_attr_setdetachstate( pthread_attr_t *attr,
                                int detachstate );
```

Library

rtxLib.a

Description

pthread_attr_setdetachstate() sets the **detachstate** attribute in the thread attributes object pointed by *attr*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

The value specified by *detachstate* must be one of the following constants, which are defined in the file **<pthread.h>**:

- PTHREAD_CREATE_JOINABLE
- PTHREAD_CREATE_DETACHED

If successful, **pthread_attr_setdetachstate()** returns 0. Otherwise, **pthread_attr_setdetachstate()** returns an error number.

Errors

[EINVAL]	The value of <i>detachstate</i> is not valid or the object pointed to by <i>attr</i> does not exist.
----------	--

Example

See **mq_open()**, p. 441.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §16.1.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

pthread_attr_setfp_np()

Synopsis

```
#include <pthread.h>

int pthread_attr_setfp_np( pthread_attr_t *attr, int fpflag );
```

Library

rtxLib.a

Description

pthread_attr_setfp_np() sets the floating point availability flag in the thread attributes object pointed to by *attr*.

If *fpflag* is set to **PTHREAD_FP_AVAILABLE_NP**, the thread created using the thread attributes object can access processor floating point registers (FPRs) and execute floating point instructions.

If *fpflag* is set to **PTHREAD_FP_NOTAVAILABLE_NP**, the thread created using the thread attributes object cannot access the processor floating point registers. An attempt by the thread to execute a floating point instruction causes the thread to be suspended.

PTHREAD_FP_AVAILABLE_NP and **PTHREAD_FP_NOTAVAILABLE_NP** are defined in the file **<pthread.h>**.

If successful, **pthread_attr_setfp_np()** returns 0. Otherwise, **pthread_attr_setfp_np()** returns an error number.

Errors

EINVAL	The object pointed to by <i>attr</i> does not exist.
ENOTSUP	<i>fpflag</i> is not equal to PTHREAD_FP_AVAILABLE_NP or PTHREAD_FP_NOTAVAILABLE_NP .

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **pthread_attr_getfp_np()**, p. 509
- **pthread_attr_init()**, p. 519()
- **pthread_create()**, p. 552()

Synopsis

```
#include <pthread.h>

int pthread_attr_setinheritsched( pthread_attr_t *attr,
                                int inherit );
```

Library

rtxLib.a

Description

pthread_attr_setinheritsched() sets the **inheritsched** attribute in the thread attributes object pointed to by *attr*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

The value specified by *inherit* must be one of the following constants, which are defined in the file **<pthread.h>**:

- PTHREAD_EXPLICIT_SCHED
- PTHREAD_INHERIT_SCHED

If successful, **pthread_attr_setinheritsched()** returns 0. Otherwise, **pthread_attr_setinheritsched()** returns an error number.

Errors

[EINVAL]	The object pointed to by <i>attr</i> does not exist.
[ENOTSUP]	The value of <i>inherit</i> is not valid.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.3.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

pthread_attr_setschedparam()

Synopsis

```
#include <pthread.h>

int pthread_attr_setschedparam( pthread_attr_t *attr,
                               const struct sched_param *param );
```

Library

rtxLib.a

Description

pthread_attr_setschedparam() sets the scheduling parameter attributes in the thread attributes object pointed to by *attr*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

The value of *param->sched_priority* must be an integer value in the inclusive range of **PTHREAD_PRIO_MIN_NP** through **PTHREAD_PRIO_MAX_NP**. These constants are defined in the file **<pthread.h>**.

If successful, **pthread_attr_setschedparam()** returns 0. Otherwise, **pthread_attr_setschedparam()** returns an error number.

Errors

[EINVAL]	The object pointed to by <i>attr</i> does not exist.
[ENOTSUP]	The value of <i>param->sched_priority</i> is not valid.

Example

The following example creates a thread with the smallest stack size, a process contention scope, and the lowest priority using a global attribute that was previously initialized. The *mem_usage* function is the example for **memheap_query()** on page 410.

```
#include <pthread.h>
#include <stddef.h>

void *mem_usage(void *file_name)

pthread_attr_t global_thread_attr;

pthread_t start_memory_usage_log(char *name)
{
    size_t stack;
    pthread_t tid;
    int scope;
    struct sched_param sparam;
    /* initialize an attributes object for the thread */
```

pthread_attr_setschedparam()

```
if (pthread_attr_init(&global_thread_attr) != 0)
    return (-1);
/* check to see what current stack size is */
pthread_attr_getstacksize(&global_thread_attr, &stack);
if(stack > sysconf(_SC_THREAD_STACK_MIN))
    pthread_attr_setstacksize(&global_thread_attr,
        sysconf(_SC_THREAD_STACK_MIN));
/* Check to see what current contention scope is */
pthread_attr_getscope(&global_thread_attr, &scope);
if(scope != PTHREAD_SCOPE_PROCESS)
    pthread_attr_setscope(&global_thread_attr,
        PTHREAD_SCOPE_PROCESS);
/* check to see what current priority is */
pthread_attr_getschedparam(&global_thread_attr, &sparam);
if(sparam.sched_priority) > PTHREAD_PRIO_MIN_NP) {
    sparam.sched_priority=PTHREAD_PRIO_MIN_NP;
    pthread_attr_setschedparam(&global_thread_attr, &sparam);
}
/* create the mem_usage() thread */
pthread_create(&tid, &global_thread_attr, mem_usage, (void *)name);
return(tid);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.3.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

pthread_attr_setschedpolicy()

Synopsis

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_setschedpolicy( pthread_attr_t *attr,
                                int policy );
```

Library

rtxLib.a

Description

pthread_attr_setschedpolicy() sets the **schedpolicy** attribute in the thread attributes object pointed to by *attr*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

The value specified by *policy* must be one of the following constants, which are defined in the file **<sched.h>**:

- SCHED_FIFO
- SCHED_RR
- SCHED_OTHER

SCHED_FIFO, SCHED_RR and SCHED_OTHER can be modified by ORing with SCHED_NOPREEMPT flag.

If successful, **pthread_attr_setschedpolicy()** returns 0. Otherwise, **pthread_attr_setschedpolicy()** returns an error number.

Errors

[EINVAL]	The object pointed to by <i>attr</i> does not exist.
[ENOTSUP]	The value specified by <i>policy</i> is not valid.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.3.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

Synopsis

```
#include <pthread.h>

int pthread_attr_setscope( pthread_attr_t *attr,
                          int contentionscope );
```

Library

rtxLib.a

Description

pthread_attr_setscope() sets the **contentionscope** attribute in the thread attributes object pointed to by *attr*.

The thread attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

The value specified by *contentionscope* must be one of the following constants, which are defined in the file **<pthread.h>**:

- PTHREAD_SCOPE_SYSTEM
- PTHREAD_SCOPE_PROCESS

If successful, **pthread_attr_setscope()** returns 0. Otherwise, **pthread_attr_setscope()** returns an error number.

Errors

[EINVAL]	The object pointed to by <i>attr</i> does not exist.
[ENOTSUP]	The value specified by <i>contentionscope</i> is not valid.

Example

See **pthread_attr_setschedparam()**, p. 524.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.3.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

pthread_attr_setstackaddr()

Synopsis

```
#include <pthread.h>

int pthread_attr_setstackaddr( pthread_attr_t *attr,
                              void *stackaddr );
```

Library

rtxLib.a

Description

pthread_attr_setstackaddr() sets the **stackaddr** attribute in the thread attributes object pointed to by *attr*.

pthread_attr_setstackaddr() sets the thread stack address to the result of the following calculation:

$\text{stackaddr} + \text{stacksize} - (\text{minimum stack frame})$

The minimum stack frame for the IBM XL C Compiler is 56 bytes. The minimum stack frame for the IBM High C/C++ compiler is 8 bytes.

The *attr* object must be initialized by **pthread_attr_init()** or unpredictable results may occur. The stack address will be adjusted to a full-word boundary.

If successful, **pthread_attr_setstackaddr()** returns 0. Otherwise, **pthread_attr_setstackaddr()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Example

See **pthread_attr_setschedparam()**, p. 524.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §16.1.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552

Synopsis

```
#include <pthread.h>

int pthread_attr_setstacksize( pthread_attr_t *attr,
                               size_t stacksize );
```

Library

rtxLib.a

Description

pthread_attr_setstacksize() sets the **stacksize** attribute, measured in bytes, in the thread attributes object pointed to by *attr*.

The value of *stacksize* must be an integer greater than the value of **PTHREAD_STACK_MIN**, which is a macro defined in **<pthread.h>**.

The thread attributes object must be initialized by **pthread_attr_init()** or unpredictable results may occur. *stacksize* will be adjusted to a multiple of 16 bytes.

If successful, **pthread_attr_setstacksize()** returns 0. Otherwise, **pthread_attr_setstacksize()** returns an error number.

Errors

[EINVAL]	The value of <i>stacksize</i> is not valid or the object pointed to by <i>attr</i> does not exist.
----------	--

Example

See **pthread_attr_setschedparam()**, p. 524.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §16.1.1
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552
- **sysconf()**, p. 841

pthread_attr_settg_np()

Synopsis

```
#include <pthread.h>

int pthread_attr_settg_np( pthread_attr_t *attr,
                          unsigned long tg );
```

Library

rtxLib.a

Description

pthread_attr_settg_np() sets the **tg** attribute in the thread attributes object pointed to by *attr*.

The value of *tg* must be **PTHREAD_CURRENT_TG_NP**, which specifies that a new thread will be created in the same thread group as the current thread, or **PTHREAD_NEW_TG_NP**, which specifies that a new thread will be created in a new thread group, or **PTHREAD_SET_TG_NP**, which specifies that a new thread will be created in the thread group specified in *tgid* field of the thread attributes object. The thread group must exist when the new thread is created.

Only **PTHREAD_PRIVILEGED_NP** threads can create threads with the **tg** attribute set to **PTHREAD_SET_TG_NP**.

The thread attributes object must be initialized by **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_settg_np()** returns 0. Otherwise, **pthread_attr_settg_np()** returns an error number.

Errors

[EINVAL]	The value of <i>tg</i> is not valid or the object pointed to by <i>attr</i> does not exist.
----------	---

Example

In the following example, thread1x is created in a new thread group. OS Open determines the thread group ID.

```
#include <pthread.h>

int test1()
{
    pthread_attr_t attr;
    int rc;
    pthread_t t1;
    rc = pthread_attr_init(&attr);
    rc = pthread_attr_settg_np(&attr, PTHREAD_NEW_TG_NP);
    rc = pthread_create(&t1, &attr, thread1x, (void *) 0);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes, unless PTHREAD_SET_TG_NP is specified

References

- **pthread_attr_gettg_np()**, p. 517
- **pthread_attr_gettgid_np()**, p. 518
- **pthread_attr_init()**, p. 519
- **pthread_attr_settgid_np()**, p. 532
- **pthread_create()**, p. 552

pthread_attr_settgid_np()

Synopsis

```
#include <pthread.h>

int pthread_attr_settgid_np( pthread_attr_t *attr,
                             tg_t tgid );
```

Library

rtxLib.a

Description

pthread_attr_settgid_np() sets the **tgid** attribute in the thread attributes object pointed to by *attr*. This value is used when the *tg* attribute is set to **PTHREAD_SET_TG_NP**.

tgid should contain a valid thread group ID, but its validity is checked only during the subsequent **pthread_create()** call.

The thread attributes object must be initialized by **pthread_attr_init()** or unpredictable results may occur.

If successful, **pthread_attr_settgid_np()** returns 0. Otherwise, **pthread_attr_settgid_np()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **pthread_attr_gettg_np()**, p. 517
- **pthread_attr_gettgid_np()**, p. 518
- **pthread_attr_init()**, p. 519
- **pthread_attr_settg_np()**, p. 530
- **pthread_attr_settgid_np()**, p. 532
- **pthread_create()**, p. 552
- **pthread_tgcreate_np()**, p. 598

Synopsis

```
#include <pthread.h>
int pthread_cancel( pthread_t thread );
```

Library

rtxLib.a

Description

pthread_cancel() requests cancelation of *thread*.

If successful, **pthread_cancel()** returns 0. Otherwise, **pthread_cancel()** returns an error number.

Errors

[EINVAL] *thread* does not refer to an existing thread.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §18.2.1
- **pthread_cleanup_pop()**, p. 534
- **pthread_cleanup_push()**, p. 535

pthread_cleanup_pop()

Synopsis

```
#include <pthread.h>

void pthread_cleanup_pop( int execute );
```

Library

None. This is a macro defined in `<pthread.h>`.

Description

pthread_cleanup_pop() removes the routine at the top of the calling thread's cleanup stack and executes it if *execute* is nonzero.

pthread_cleanup_pop() completes a **pthread_cleanup_push()**. The call to **pthread_cleanup_pop()** must follow a call to **pthread_cleanup_push()** within the same block.

pthread_cleanup_pop() returns nothing.

Errors

None.

Example

See **pthread_cleanup_push()**, p. 535.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §18.2.3
- **pthread_cancel()**, p. 533
- **pthread_cleanup_push()**, p. 535

Synopsis

```
#include <pthread.h>

void pthread_cleanup_push( void (*routine)(void *),
                           void *arg );
```

Library

None. This macro is defined in `<pthread.h>`.

Description

pthread_cleanup_push() pushes the cleanup routine pointed to by *routine* onto the calling thread's cleanup stack.

The cleanup routine is popped from the stack and executed with the argument pointed to by *arg* when the thread exits using **pthread_exit()**, acts on a cancelation request, or calls **pthread_cleanup_pop()** with a non-zero *execute* argument.

To complete this macro, **pthread_cleanup_push()** must be followed by a call to **pthread_cleanup_pop()** within the same block. Multiple routines can be pushed on the stack.

pthread_cleanup_push() returns nothing.

Errors

None.

Example

The following example implements the writer's lock of a cancelable, writer's-priority reader's/writer's lock.

```
#include <pthread.h>

typedef struct
{
    pthread_mutex_t lock;
    pthread_cond_t rcond;
    pthread_cond_t wcond;
    int lock_count;    /* < 0, held by writer      */
                      /* > 0, held by reader      */
                      /* = 0, held by nobody     */
    int waiting_writers; /* writers waiting for lock count */
} rwlock;

void decWwrlsemutex(void *arg)
{
    rwlock *l = (rwlock *)arg;
```

pthread_cleanup_push()

```
/* If there are no writers waiting and lock is held by a reader */
if((!l->waiting_writers == 0) && (l->lock_count >= 0))
    /* Wake up all threads blocked on writer's condition */
    pthread_cond_broadcast(&l->wcond);
/* Unlock the mutex */
pthread_mutex_unlock(&l->lock);
}
void getwriters(rwlock *l)
{
    /* Lock the mutex */
    pthread_mutex_lock(&l->lock);
    /* increment the waiting writers' count */
    l->waiting_writers++;
    /* push cleanup routine to release the lock */
    pthread_cleanup_push(decVWrlsemutex, l);
    /* while some thread holds the lock wait on writer's condition */
    while(l->lock_count != 0)
        pthread_cond_wait(&l->wcond, &l->lock);
    /* set lock count to state lock is held by a writer */
    l->lock_count = -1;
    /* execute clean up routine to release the lock */
    pthread_cleanup_pop(1);
}
void rlsewriters(rwlock *l)
{
    /* Lock the mutex */
    pthread_mutex_lock(&l->lock);
    /* Set lock count to state no one holds the lock */
    l->lock_count = 0;
    /* If no writers are waiting on the lock */
    if(l->waiting_writers == 0)
        /* Wake all threads blocked on reader's condition */
        pthread_cond_broadcast(&l->rcond);
    else
        /* Signal next thread blocked on writer's condition */
        pthread_cond_signal(&l->wcond);
    /* Unlock the mutex */
    pthread_mutex_unlock(&l->lock);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §18.2.3
- **pthread_cancel()**, p. 533
- **pthread_cleanup_pop()**, p. 534

pthread_cond_broadcast()

Synopsis

```
#include <pthread.h>
int pthread_cond_broadcast( pthread_cond_t *cond );
```

Library

rtxLib.a

Description

pthread_cond_broadcast() unblocks all threads blocked on the condition variable pointed to by *cond*.

pthread_cond_broadcast() has no effect if no threads are blocked on *cond*. If multiple threads are blocked on the condition variable, the scheduling policy determines the order in which the threads are unblocked.

Any thread unblocked by **pthread_cond_broadcast()** resumes only after the thread reacquires the mutex with which the thread called **pthread_cond_wait()** or **pthread_cond_timedwait()**. Unblocked threads contend for the mutex according to the scheduling policy, as if each had called **pthread_mutex_lock()**.

If successful, **pthread_cond_broadcast()** returns 0. Otherwise, **pthread_cond_broadcast()** returns an error number.

Errors

[EINVAL]	The condition variable pointed to by <i>cond</i> does not exist.
[ENOMEM]	Insufficient memory is available.

Example

See **pthread_cleanup_push()**, p. 535.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §11.4.3
- **pthread_cond_signal()**, p. 543
- **pthread_cond_timedwait()**, p. 544
- **pthread_cond_wait()**, p. 546

Synopsis

```
#include <pthread.h>

int pthread_cond_destroy( pthread_cond_t *cond );
```

Library

rtxLib.a

Description

pthread_cond_destroy() destroys the condition variable pointed to by *cond*.

Using an identifier that points to a destroyed condition variable produces erroneous results.

If successful, **pthread_cond_destroy()** returns 0. Otherwise, **pthread_cond_destroy()** returns an error number.

Errors

[EINVAL]	The condition variable pointed to by <i>cond</i> does not exist.
[EBUSY]	One or more threads are currently blocked on the condition variable pointed to by <i>cond</i> .

Example

See **pthread_cond_init()**, p. 540.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §11.4.2
- **pthread_cond_init()**, p. 540

pthread_cond_init()

Synopsis

```
#include <pthread.h>
int pthread_cond_init( pthread_cond_t *cond,
                      pthread_condattr_t *attr );
```

Library

rtxLib.a

Description

pthread_cond_init() creates a condition variable and stores it in the object pointed to by *cond*.

The condition variable is initialized with attributes pointed to by *attr*. If these attributes are modified, the condition variable attributes are not affected. If *attr* is NULL, the default condition variable attributes are used.

If successful, **pthread_cond_init()** returns 0. Otherwise, **pthread_cond_init()** returns an error number.

Errors

[EAGAIN]	Insufficient resources (other than memory).
[EBUSY]	The object pointed to by <i>cond</i> contains an existing condition variable.
[ENOMEM]	Insufficient memory.
[EINVAL]	The object pointed to by <i>attr</i> does not exist.

Example

The following example suspends a thread on a condition variable. A second function resumes a thread.

```
#include <pthread.h>
#include <stddef.h>

struct susp
{
    struct susp *next; /* next structure in linked list */
    pthread_t who; /* thread ID of thread suspended */
    pthread_cond_t w; /* condition variable */
    pthread_mutex_t m; /* mutex */
} *list = NULL;

void suspend_self(void)
{
    struct susp *s;
    s = (struct susp *)malloc(sizeof(struct susp));
```

```
/* Create mutex for this thread */
pthread_mutex_init(&s->m, NULL);
/* Create a condition variable */
pthread_cond_init(&s->w, NULL);
/* Initialize remaining fields in the structure */
s->who = pthread_self();
s->next = list;
list = s;
/* Lock mutex */
pthread_mutex_lock(&s->m);
/* Wait on the condition variable */
pthread_cond_wait(&s->w, &s->m);
/* pthread_cond_wait returns when a resume() is called for */
/* this function. Destroy the condition variable and mutex */
/* because they are no longer needed. */
pthread_cond_destroy(&s->w);
pthread_mutex_unlock(&s->m);
pthread_mutex_destroy(&s->m);
free(s);
}

void resume(pthread_t t)
{
    struct susp *p, *q;

    /* Look for the thread in the list */
    q = list;
    for(p = list; p != NULL; p = p->next)
    {
        /* If thread is found */
        if(pthread_equal(t, p->who))
        {
            if(q == p)
                list = p->next;
            q->next = p->next;
            pthread_mutex_lock(&p->m);
            /* Wake the thread blocked on this condition variable */
            pthread_cond_signal(&p->w);
            pthread_mutex_unlock(&p->m);
            break;
        }
        q = p;
    }
}
```

pthread_cond_init()

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §11.4.2
- **pthread_cond_destroy()**, p. 539
- **pthread_condattr_destroy()**, p. 548

Synopsis

```
#include <pthread.h>

int pthread_cond_signal( pthread_cond_t *cond );
```

Library

rtxLib.a

Description

pthread_cond_signal() unblocks a thread blocked on the condition variable pointed to by *cond*.

If more than one thread is blocked on the condition variable, the scheduling policy determines which thread is unblocked. This call has no effect if no threads are blocked on the condition variable.

A thread unblocked by **pthread_cond_signal()** resumes only after the thread reacquires the mutex used to call **pthread_cond_wait()** or **pthread_cond_timedwait()**. The unblocked thread contends for the mutex according to the scheduling policy, as if it had called **pthread_mutex_lock()**.

If successful, **pthread_cond_signal()** returns 0. Otherwise, **pthread_cond_signal()** returns an error number.

Errors

[EINVAL]	The condition variable pointed to by <i>cond</i> does not exist.
[ENOMEM]	Insufficient memory.

Example

See **pthread_cond_init()**, p. 540.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §11.4.3
- **pthread_cond_broadcast()**, p. 538
- **pthread_cond_timedwait()**, p. 544
- **pthread_cond_wait()**, p. 546

pthread_cond_timedwait()

Synopsis

```
#include <time.h>
#include <pthread.h>

int pthread_cond_timedwait( pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

Library

rtxLib.a

Description

pthread_cond_timedwait() blocks the calling thread on a condition variable pointed to by *cond*.

pthread_cond_timedwait() stops waiting if the time pointed to by *abstime* (absolute time) passes.

pthread_cond_timedwait() must be called with the object pointed to by *mutex* locked by the calling thread. The function releases the mutex and waits on the condition variable. When the wait is satisfied, the mutex is reacquired and **pthread_cond_timedwait()** returns.

This function creates a temporary binding between the condition variable and the mutex. If two or more threads attempt to block on the same condition variable at the same time, they must specify the same mutex when calling **pthread_cond_timedwait()** or **pthread_cond_wait()**.

If successful, **pthread_cond_timedwait()** returns 0. Otherwise, **pthread_cond_timedwait()** returns an error number.

Errors

[EINVAL]	The condition variable pointed to by <i>cond</i> does not exist; the mutex pointed to by <i>mutex</i> does not exist, or is not locked by the calling thread, or differs from the mutex specified by another thread blocked on the condition variable.
[EINTR]	An unblocked signal is pending on the system or the thread.
[ENOMEM]	Insufficient memory.
[ETIMEDOUT]	The time pointed to by <i>abstime</i> passed.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §11.4.4
- **pthread_cond_broadcast()**, p. 538
- **pthread_cond_signal()**, p. 543
- **pthread_cond_wait()**, p. 546

pthread_cond_wait()

Synopsis

```
#include <pthread.h>

int pthread_cond_wait( pthread_cond_t *cond,
                      pthread_mutex_t *mutex );
```

Library

rtxLib.a

Description

pthread_cond_wait() blocks the calling thread on a condition variable pointed to by *cond*.

pthread_cond_wait() must be called with the mutex pointed to by *mutex* locked by the calling thread. The function releases the mutex and waits on the condition variable. When the wait is satisfied, the mutex is reacquired and **pthread_cond_wait()** returns.

pthread_cond_wait() creates a temporary binding between the condition variable and the mutex. If two or more threads attempt to block on the same condition variable at the same time, they must specify the same mutex when calling **pthread_cond_timedwait()** or **pthread_cond_wait()**.

If successful, **pthread_cond_wait()** returns 0. Otherwise, **pthread_cond_wait()** returns an error number.

Errors

[EINVAL]	The condition variable pointed to by <i>cond</i> does not exist; the mutex pointed to by <i>mutex</i> does not exist, or is not locked by the calling thread, or differs from the mutex specified by another thread currently blocked on the condition variable.
[EINTR]	An unblocked signal is pending on the system or the thread.
[ENOMEM]	Insufficient memory.

Example

See **pthread_cond_init()**, p. 540.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §11.4.4
- **pthread_cond_broadcast()**, p. 538
- **pthread_cond_signal()**, p. 543
- **pthread_cond_wait()**, p. 546

pthread_condattr_destroy()

Synopsis

```
#include <pthread.h>
int pthread_condattr_destroy( pthread_condattr_t *attr );
```

Library

rtxLib.a

Description

pthread_condattr_destroy() destroys the condition attributes object pointed to by *attr* by setting the object to 0.

The condition attributes object must have been initialized using **pthread_condattr_init()** or unpredictable results may occur. Using an identifier that points to a destroyed condition attributes object produces erroneous results.

If successful, **pthread_condattr_destroy()** returns 0. Otherwise, **pthread_condattr_destroy()** sets an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §11.4.1
- **pthread_condattr_init()**, p. 550

Synopsis

```
#include <pthread.h>

int pthread_condattr_getpshared( const pthread_attr_t *attr,
                                int *pshared);
```

Library

rtxLib.a

Description

pthread_condattr_getpshared() gets the *pshared* parameter from the condition attributes object pointed to by *attr*.

The condition attributes object must be initialized by **pthread_condattr_init()** or unpredictable results may occur.

If successful, **pthread_condattr_getpshared()** returns 0. Otherwise, **pthread_condattr_getpshared()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **pthread_condattr_destroy()**, p. 548
- **pthread_condattr_init()**, p. 550

pthread_condattr_init()

Synopsis

```
#include <pthread.h>
int pthread_condattr_init( pthread_condattr_t *attr );
```

Library

rtxLib.a

Description

pthread_condattr_init() creates and initializes a condition attributes object and stores an identifier for the newly created object in the field pointed to by *attr*.

Each attribute in the newly created object is set to its system default value. Currently, the object contains no individual attributes; **pthread_condattr_init()** is provided only for portability.

pthread_condattr_init() always returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §11.4.1
- **pthread_cond_init()**, p. 540
- **pthread_condattr_destroy()**, p. 548

Synopsis

```
#include <pthread.h>

int pthread_condattr_setpshared( pthread_condattr_t *attr,
                                int pshared );
```

Library

rtxLib.a

Description

pthread_condattr_setpshared() sets the process shared attribute to the value of the *pshared* parameter in the condition attributes object pointed to by *attr*.

The value of *pshared* must be either **PTHREAD_PROCESS_SHARED**, which specifies that the new condition variable will be owned by the kernel thread group, or **PTHREAD_PROCESS_PRIVATE**, which specifies that the new condition variable will be owned by the creating thread group and will be destroyed when the creating thread group is destroyed.

The condition attributes object must be initialized by **pthread_condattr_init()** or unpredictable results may occur.

If successful, **pthread_condattr_setpshared()** returns 0. Otherwise, **pthread_condattr_setpshared()** returns an error number.

Errors

[EINVAL]	The value of <i>pshared</i> is not valid or the object pointed to by <i>attr</i> does not exist.
----------	--

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **pthread_condattr_destroy()**, p. 548
- **pthread_condattr_getpshared()**, p. 549
- **pthread_condattr_init()**, p. 550
- **pthread_tgdestroy_np()**, p. 599

pthread_create()

Synopsis

```
#include <pthread.h>

int pthread_create( pthread_t *thread,
                   const pthread_attr_t *attr,
                   void * (*start_routine)(void *),
                   void *arg );
```

Library

rtxLib.a

Description

pthread_create() creates and initializes a new thread and stores its identifier in the field pointed to by *thread*.

The thread is initialized with attributes in a thread attributes object pointed to by *attr*. If the attributes pointed to by *attr* are later modified, the threads attributes are not affected. If *attr* is NULL, default thread attributes are used. Otherwise, the threads attributes object must be initialized using **pthread_attr_init()** or unpredictable results may occur.

When the thread is created, it executes the routine pointed to by *start_routine* if the routine exists. *arg* is the sole argument. If the routine returns, the effect is as if **pthread_exit()** was called implicitly, using the return value of the specified routine as the exit status.

If successful, **pthread_create()** returns 0. Otherwise, **pthread_create()** returns an error number and the field pointed to by *thread* is unchanged.

Errors

[EAGAIN]	Insufficient memory.
[EINVAL]	The object pointed to by <i>attr</i> does not exist.

Example

See **mq_open()**, p. 441.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §16.1.2
- **pthread_attr_init()**, p. 519
- **pthread_exit()**, p. 557

pthread_detach()

Synopsis

```
#include <pthread.h>
int pthread_detach( pthread_t thread );
```

Library

rtxLib.a

Description

pthread_detach() releases storage allocated for *thread* so the storage can be reclaimed when *thread* terminates. **pthread_detach()** does not cause *thread* to terminate.

If successful, **pthread_detach()** returns 0. Otherwise, **pthread_detach()** returns an error number and the thread pointed to by *thread* is unchanged.

Errors

[EINVAL]	The thread specified by <i>thread</i> does not exist.
[ESRCH]	The thread specified by <i>thread</i> is already detached.

Examples

See **mq_open()**, p. 441

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **pthread_join()**, p. 563

Synopsis

```
#include <pthread.h>
int pthread_equal( pthread_t t1,
                  pthread_t t2 );
```

Library

rtxLib.a

Description

pthread_equal() compares thread IDs *t1* and *t2*.

pthread_equal() does not check whether *t1* and *t2* identify existing threads.

pthread_equal() returns nonzero if *t1* and *t2* are equal. Otherwise, **pthread_equal()** returns 0.

Errors

None.

Example

See **pthread_cond_init()**, p. 540.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §16.1.7

pthread_errno_np()

Synopsis

```
#include <errno.h>
int *pthread_errno_np( void );
```

Library

rtxLib.a

Description

pthread_errno_np() returns a pointer to the calling thread's *errno* value.

The *errno* value is set by many C and system functions to indicate error conditions. **pthread_errno_np()** returns a pointer to the thread's *errno* value and thus can be used to both retrieve and set the value.

The macro **errno**, defined in the file **<errno.h>**, calls **pthread_errno_np()** and allows *errno* to be treated as a simple integer variable from an application.

Errors

None.

Example

See **sem_open()**, p. 700.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <pthread.h>

void pthread_exit( void *value_ptr );
```

Library

rtxLib.a

Description

pthread_exit() terminates the calling thread and makes *value_ptr* available to any successful join with the terminating thread.

An implicit call to **pthread_exit()** is made when a thread returns from the start routine that was used to create it, with the function's return value serving as the thread's exit status.

Any cancelation cleanup handlers on the cleanup handler stack are popped and executed. Destructor calls are repeated as needed to reclaim all thread-specific data at least **{PTHREAD_DESTRUCTOR_ITERATIONS}** times in case a destructor function causes the creation of more thread-specific data. Thread termination does not release any resources visible to an application, including, but not limited to, mutexes, condition variables, attributes objects and storage.

If the **detachstate** attribute of the thread is **PTHREAD_CREATE_DETACHED**, the terminated thread and its stack are freed. Otherwise, the terminated thread and its stack are not freed unless a **pthread_join()** call specified the terminated thread.

The behavior of **pthread_exit()** is undefined if called from a cleanup handler or key destructor function.

pthread_exit() does not return to its caller.

Errors

None.

Example

See **getchar_unlocked()**, p. 258.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

pthread_exit()

References

- IEEE Draft Standard 1003.4a/D8, §16.1.5
- **pthread_create()**, p. 552
- **pthread_join()**, p. 563

Synopsis

```
#include <pthread.h>

void pthread_getpr_np( unsigned long *privilege);
```

Library

rtxLib.a

Description

pthread_attr_getpr_np() gets the privilege status of the current thread and stores it in the variable pointed to by *privilege*.

Privilege can be either **PTHREAD_PRIVILEGED_NP** or **PTHREAD_APPLICATION_NP**.

During system call thread privilege is temporarily set to **PTHREAD_PRIVILEGED_NP**. On exit from the system call routine thread privilege is restored to its original value.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

References

- **pthread_attr_gettg_np()**, p. 517
- **pthread_attr_init()**, p. 519
- **pthread_attr_settg_np()**, p. 530
- **pthread_create()**, p. 552
- **pthread_tgcreate_np()**, p. 598

pthread_getschedparam()

Synopsis

```
#include <pthread.h>

int pthread_getschedparam( pthread_t thread,
                           int *policy,
                           struct sched_param *param );
```

Library

rtxLib.a

Description

pthread_getschedparam() retrieves the scheduling policy and the scheduling parameters of the thread identified by *thread*, and stores those values in locations pointed to by *policy* and *param*, respectively.

For **SCHED_FIFO** and **SCHED_RR**, the valid values for the OS Open implementation, the only scheduling parameter is priority.

If successful, **pthread_getschedparam()** returns 0. Otherwise, **pthread_getschedparam()** returns an error number.

Errors

[ESRCH] *thread* does not refer to an existing thread.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.3.2
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552
- **pthread_setschedparam()**, p. 592

Synopsis

```
#include <pthread.h>

void *pthread_getspecific( pthread_key_t key );
```

Library

rtxLib.a

Description

pthread_getspecific() gets the value currently bound to the specified *key* on behalf of the calling thread.

The effect of calling this function either explicitly or implicitly from a thread specific data destructor function is undefined.

If successful, **pthread_getspecific()** returns the thread-specific data value associated with the given *key*. If no thread-specific data value is associated with the *key*, the value NULL is returned.

Errors

None.

Example

See **pthread_key_create()**, p. 564.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §17.1.2
- **pthread_key_create()**, p. 564
- **pthread_setspecific()**, p. 593

pthread_getstackaddr_np()

Synopsis

```
#include <pthread.h>

int pthread_getstackaddr_np( unsigned long *stack_bottom );
```

Library

rtxLib.o

Description

pthread_getstackaddr_np() retrieves stack bottom of the current thread and stores its value in the variable pointed to by *stack_bottom*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **pthread_attr_getstackaddr()**, p. 515
- **pthread_attr_getstacksize()**, p. 516
- **pthread_attr_setstackaddr()**, p. 528
- **pthread_attr_setstacksize()**, p. 529

Synopsis

```
#include <pthread.h>

int pthread_join( pthread_t thread, void **status );
```

Library

rtxLib.a

Description

pthread_join() blocks the calling thread until the target *thread* terminates.

pthread_join() returns without blocking if *thread* has terminated, or if another thread is blocked because it joined *thread*.

If *status* is not NULL, the terminating thread passes the value stored in the location referenced by *status* to **pthread_exit()**.

The results of a **pthread_join()** call on a detached target thread is undefined, as are the results of multiple simultaneous calls to **pthread_join()**.

If successful, **pthread_join()** returns 0.

Otherwise, **pthread_join()** returns an error number. The field referenced by *status* is unchanged.

Errors

[EDEADLK]	<i>thread</i> specifies the calling thread.
[EINVAL]	<i>thread</i> does not refer to a joinable thread.
[ESRCH]	<i>thread</i> is not valid.

Example

See **mq_open()**, p. 441.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §16.1.3
- **pthread_create()**, p. 552
- **pthread_exit()**, p. 557

pthread_key_create()

Synopsis

```
#include <pthread.h>

int pthread_key_create( pthread_key_t *key,
                      void (*destructor(void *)) );
```

Library

rtxLib.a

Description

pthread_key_create() creates a key visible to all threads and stores its value in an object pointed to by *key*.

Key values provided by **pthread_key_create()** locate thread-specific data. Although different threads can use the same key, the values bound to the key by **pthread_specific()** are maintained on a per-thread basis and persist for the life of the calling thread.

Upon key creation, the value NULL is associated with the new key in all active threads. Upon thread creation, the value NULL is associated with all defined keys in the new thread.

An optional destructor function may be associated with each key. At thread exit, if a key has a non-NULL destructor pointer and the thread has a non-NULL value associated with that key, the destructor function is called with the current key value as its sole argument. The order of destructor calls is not specified if more than one destructor exists for a thread.

Destructor calls are repeated as needed to reclaim all thread-specific data at least **{PTHREAD_DESTRUCTOR_ITERATIONS}** times in case a destructor function causes the creation of more thread-specific data.

If successful, **pthread_key_create()** returns 0. Otherwise, **pthread_key_create()** returns an error number.

Errors

[EAGAIN]	The number of keys created equals the maximum number of keys allowed.
[ENOMEM]	Insufficient memory.

Example

```

/*-----+
wait20() Function to wait 20 seconds.
This function uses thread-specific keys to store a unique pointer to a timer id
for each thread. When the thread terminates, a destructor function is called to
delete the timer id and free allocated memory.
/*-----+
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <pthread.h>
/* Static variable for pthread_once() */
static pthread_once_t once_control = PTHREAD_ONCE_INIT;
/* Static variable for key */
static pthread_key_t key;
/*-----+ | set timer. +-----*/
static int set_timer(itimerspec_t *v) {
    timer_t    *timeridp;          /* Timer ID pointer */
    /* Use timer id in specific key of calling thread. */
    timeridp=(timer_t *)pthread_getspecific(key);
    /* If key value is NULL, create a timer id for calling thread. */
    if (timeridp==(timer_t *)NULL) { timeridp=(timer_t *)malloc(sizeof(timer_t));
        if (timeridp==NULL)
            return -1;              /* Malloc error */
        /* Create timer id. */
        if(timer_create(CLOCK_REALTIME,NULL,timeridp)!=0)
            return -1;              /* timer create error */
        /* Set thread specific key to timer id pointer. */
        pthread_setspecific(key, (void *)timeridp);
    }
    /* Set relative timer. */
    timer_settime(*timeridp,TIMER_RELTIME_NP,v,NULL);
    return 0;
} /* end set_timer() */
/*-----+ delete timer (destructor function) +-----*/
static void delete_timer(void *value) {
    /* Delete timer pointed to by passed key value. */
    timer_delete(*(timer_t *)value);
    /* Free allocated memory. */
    free(value);
    /* Set key value to NULL. */
    pthread_setspecific(key,NULL);
} /* end delete_timer() */
/*-----+ | One time only initialization. +-----*/
static void call_once() {

```

pthread_key_create()

```
    /* Create a thread specific key, specifying delete_timer() as a
       destructor function (to be called on thread termination).    */
    pthread_key_create(&key,delete_timer);
} /* end call_once() */
/*-----+ | Wait 20 seconds. +-----*/
int wait20() {
    int sig;
    itimerspec_t v = {0};
    sigset_t set, oldset;
    /* One time only initialization */
    pthread_once(&once_control, call_once);
    /* Set 20 second timer. */
    v.it_value.tv_sec = 20;
    if (set_timer(&v)!= 0)
        return -1;
    /* Set signal mask to receive SIGALRM (timer signal). */
    sigemptyset(&set);
    sigaddset(&set,SIGALRM);
    pthread_sigmask(SIG_UNBLOCK,&set,&oldset); /* Unblock SIGALRM */
    /* Wait for SIGALRM signal */
    sigwait(&set, &sig);
    pthread_sigmask(SIG_SETMASK,&oldset,NULL); /* Restore signal mask */
    return 0;
} /* end wait20() */
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §17.1.1
- **pthread_getspecific()**, p. 561
- **pthread_setspecific()**, p. 593

Synopsis

```
#include <pthread.h>

int pthread_key_delete( pthread_key_t key );
```

Library

rtxLib.a

Description

pthread_key_delete() deletes a thread-specific data key, named by *key*, previously returned by **pthread_key_create()**.

If thread-specific data other than a NULL is associated with *key* when **pthread_key_delete()** is called, the effect is undefined. Attempts to use *key* after a call to **pthread_key_delete()** results in undefined behavior.

If successful, **pthread_key_delete()** returns 0. Otherwise, **pthread_key_delete()** returns an error number.

Errors

[EINVAL] *key* is not valid.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §17.1.3
- **pthread_getspecific()**, p. 561
- **pthread_setspecific()**, p. 593

pthread_kill()

Synopsis

```
#include <pthread.h>
#include <signal.h>
int pthread_kill( pthread_t thread, int sig );
```

Library

rtxLib.a

Description

pthread_kill() directs the signal *sig* to be asynchronously delivered to the thread pointed to by *thread*.

If *sig* is 0, error checking is performed, but no signal is sent.

If successful, **pthread_kill()** returns 0. Otherwise, **pthread_kill()** returns an error number.

Errors

[EINVAL]	The value of <i>sig</i> is not valid or is an unsupported signal number.
[ESRCH]	<i>thread</i> is not valid.

Example

See **sigwait()**, p. 751.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §3.3.11
- **kill()**, p. 350
- **raise()**, p. 630

Synopsis

```
#include <pthread.h>

int pthread_mutex_destroy( pthread_mutex_t *mutex );
```

Library

rtxLib.a

Description

pthread_mutex_destroy() destroys the mutex pointed to by *mutex*.

Using an identifier that points to a destroyed mutex produces erroneous results.

If successful, **pthread_mutex_destroy()** returns 0. Otherwise, **pthread_mutex_destroy()** returns an error number.

Errors

[EBUSY]	<i>mutex</i> points to a mutex that is locked or referenced (for example, while being used in pthread_cond_wait() or pthread_cond_timedwait()) by another thread.
[EINVAL]	The mutex pointed to by <i>mutex</i> does not exist.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §11.3.2
- **pthread_mutex_init()**, p. 571

pthread_mutex_getprioceiling()

Synopsis

```
#include <pthread.h>

int pthread_mutex_getprioceiling( pthread_mutex_t *mutex,
                                  int *prioceiling );
```

Library

rtxLib.a

Description

pthread_mutex_getprioceiling() gets the **prioceiling** attribute for the mutex identified by *mutex*.

pthread_mutex_getprioceiling() sets the location pointed to by *prioceiling* to an integer value in the inclusive range of the constants **PTHREAD_PRIO_MIN_NP** through **PTHREAD_PRIO_MAX_NP**. These constants are defined in the file **<pthread.h>**.

If successful, **pthread_mutex_getprioceiling()** returns 0. Otherwise, **pthread_mutex_getprioceiling()** returns an error number.

Errors

[EINVAL]	<i>prioceiling</i> is out of range.
[EPERM]	The application program is not sufficiently privileged.
[ESRCH]	The mutex pointed to by <i>mutex</i> does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.4.2
- **pthread_mutex_init()**, p. 571
- **pthread_mutexattr_setprioceiling()**, p. 584

Synopsis

```
#include <pthread.h>

int pthread_mutex_init( pthread_mutex_t *mutex,
                        pthread_mutexattr_t *attr );
```

Library

rtxLib.a

Description

pthread_mutex_init() creates and initializes a mutex and stores its identifier in the object pointed to by *mutex*.

The mutex is initialized with attributes specified by *attr*. If the attributes specified by *attr* are later modified, the mutex attributes are not affected. If *attr* is NULL the default mutex attributes are used. Otherwise, the *attr* object must be initialized using **pthread_mutexattr_init()** or unpredictable results may occur.

If successful, **pthread_mutex_init()** returns 0. Otherwise, **pthread_mutex_init()** returns an error number.

Errors

[ENOMEM]	Insufficient memory.
[EINVAL]	The object pointed to by <i>attr</i> does not exist.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §11.3.2
- **pthread_mutexattr_init()**, p. 581
- **pthread_mutex_destroy()**, p. 569

pthread_mutex_lock()

Synopsis

```
#include <pthread.h>
int pthread_mutex_lock( pthread_mutex_t *mutex );
```

Library

rtxLib.a

Description

pthread_mutex_lock() locks the mutex specified by *mutex*.

If the mutex is already locked, the calling thread is blocked until the mutex is unlocked. This operation returns with the mutex in the locked state.

If successful, **pthread_mutex_lock()** returns 0. Otherwise, **pthread_mutex_lock()** returns an error number.

Errors

[EDEADLK]	The mutex pointed to by <i>mutex</i> is already locked by the calling thread.
[EINVAL]	The mutex pointed to by <i>mutex</i> does not exist.
[ENOMEM]	Insufficient memory.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §11.3.2
- **pthread_mutex_trylock()**, p. 575
- **pthread_mutex_unlock()**, p. 576

Synopsis

```
#include <pthread.h>

int pthread_mutex_setprioceiling( pthread_mutex_t *mutex,
                                int prioceiling,
                                int *old_ceiling );
```

Library

rtxLib.a

Description

pthread_mutex_setprioceiling() changes the priority of the mutex specified by *mutex* to the value specified by *prioceiling*.

The value specified by *prioceiling* must be an integer value in the inclusive range of **PTHREAD_PRIO_MIN_NP** through **PTHREAD_PRIO_MAX_NP**. These constants are defined in the file **<pthread.h>**.

If successful, **pthread_mutex_setprioceiling()** returns 0 and sets the location pointed to by *old_ceiling* to the prior value of the prioceiling attribute. Otherwise, **pthread_mutex_setprioceiling()** returns an error number.

Errors

[ESRCH]	The mutex pointed to by <i>mutex</i> does not exist.
[EPERM]	The application program is not sufficiently privileged.
[EINVAL]	The value specified by <i>prioceiling</i> is not valid.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §13.4.2
- **pthread_mutex_getprioceiling()**, p. 570
- **pthread_mutex_init()**, p. 571
- **pthread_mutexattr_setprioceiling()**, p. 584

pthread_mutex_timedlock()

Synopsis

```
#include <pthread.h>

int pthread_mutex_timedlock( pthread_mutex_t *mutex,
                             const struct timespec *timeout );
```

Library

rtxLib.a

Description

pthread_mutex_timedlock() locks the mutex specified by *mutex*.
pthread_mutex_timedlock() times out if time specified in *timeout* has elapsed and the thread was not able to acquire the mutex.

If the mutex is already locked, the calling thread is blocked until the mutex is unlocked or *timeout* expires. If successful, operation returns with the mutex in the locked state.

If successful, **pthread_mutex_timedlock()** returns 0. Otherwise, **pthread_mutex_timedlock()** returns an error number.

Errors

[EDEADLK]	The mutex pointed to by <i>mutex</i> is already locked by the calling thread.
[EINVAL]	The mutex pointed to by <i>mutex</i> does not exist.
[ENOMEM]	Insufficient memory.
[ETIMEDOUT]	The time specified in <i>timeout</i> has elapsed.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **pthread_mutex_lock()**, p. 572
- **pthread_mutex_trylock()**, p. 575
- **pthread_mutex_unlock()**, p. 576

Synopsis

```
#include <pthread.h>

int pthread_mutex_trylock( pthread_mutex_t *mutex );
```

Library

rtxLib.a

Description

pthread_mutex_trylock() locks the mutex pointed to by *mutex*.

If the mutex is already locked, the call returns immediately without affecting the state of the mutex.

If successful, **pthread_mutex_trylock()** returns 0. Otherwise, **pthread_mutex_trylock()** returns an error number.

Note: If *mutex* is **NO_PRIO_INHERIT**, **pthread_mutex_trylock()** is interrupt-handler safe.

Errors

[EINVAL]	The mutex pointed to by <i>mutex</i> does not exist.
[EBUSY]	The mutex pointed to by <i>mutex</i> is already locked.
[EDEADLK]	The mutex pointed to by <i>mutex</i> is already locked by the calling thread.
[ENOMEM]	Insufficient memory.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §11.3.3
- **pthread_mutex_lock()**, p. 572
- **pthread_mutex_unlock()**, p. 576

pthread_mutex_unlock()

Synopsis

```
#include <pthread.h>
int pthread_mutex_unlock( pthread_mutex_t *mutex );
```

Library

rtxLib.a

Description

pthread_mutex_unlock() unlocks the mutex pointed to by *mutex*.

The mutex must be locked by the calling thread. If more than one thread is waiting for the mutex when **pthread_mutex_unlock()** is called, the scheduling policy determines which thread will lock the mutex next.

If successful, **pthread_mutex_unlock()** returns 0. Otherwise, **pthread_mutex_unlock()** returns an error number.

Errors

[EINVAL]	The mutex pointed to by <i>mutex</i> does not exist.
[EPERM]	The thread calling pthread_mutex_unlock() does not own the mutex.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §11.3.3
- **pthread_mutex_lock()**, p. 572
- **pthread_mutex_trylock()**, p. 575

Synopsis

```
#include <pthread.h>

int pthread_mutexattr_destroy( pthread_mutexattr_t *attr );
```

Library

rtxLib.a

Description

pthread_mutexattr_destroy() destroys a mutex attributes object pointed to by *attr*.

The *attr* object must have been initialized using the **pthread_mutexattr_init()** function or unpredictable results may occur. Using an identifier that points to a destroyed mutex attributes object produces erroneous results.

If successful, **pthread_mutexattr_destroy()** returns 0. Otherwise, **pthread_mutexattr_destroy()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §11.3.1
- **pthread_mutexattr_init()**, p. 581

pthread_mutexattr_getprioceiling()

Synopsis

```
#include <pthread.h>

int pthread_mutexattr_getprioceiling( const pthread_mutexattr_t *attr,
                                     int *prioceiling );
```

Library

rtxLib.a

Description

pthread_mutexattr_getprioceiling() gets the **prioceiling** attribute from the mutex attributes object pointed to by *attr*.

pthread_mutexattr_getprioceiling() sets the location pointed to by *prioceiling* to an integer value in the inclusive range of **PTHREAD_PRIO_MIN_NP** through **PTHREAD_PRIO_MAX_NP**, which are defined in the file **<pthread.h>**

The mutex attributes object must be initialized using **pthread_mutexattr_init()** or unpredictable results may occur.

The **prioceiling** attribute has significance only when the **protocol** attribute equals **PRIO_PROTECT**. The **prioceiling** attribute defines the minimum priority at which the thread that has locked the mutex will execute.

To effectively avoid priority inversion, the priority ceiling of the mutex should be set to a priority greater than or equal to the highest priority of all the threads able to lock the mutex.

If successful, **pthread_mutexattr_getprioceiling()** returns 0. Otherwise, **pthread_mutexattr_getprioceiling()** returns an error number.

Errors

[EINVAL]	The object pointed to by <i>attr</i> does not exist.
[EPERM]	The application program is not sufficiently privileged.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.6.1
- **pthread_mutexattr_init()**, p. 581
- **pthread_mutexattr_setprioceiling()**, p. 584

Synopsis

```
#include <pthread.h>

int pthread_mutexattr_getprotocol( const pthread_mutexattr_t *attr,
                                   int *protocol );
```

Library

rtxLib.a

Description

pthread_mutexattr_getprotocol() gets the **protocol** attribute from the mutex attributes object identified by *attr*.

attr must be initialized by **pthread_mutexattr_init()** or unpredictable results may occur.

If successful, **pthread_mutexattr_getprotocol()** returns 0 and sets the location pointed to by *protocol* to one of the following values:

- NO_PRIO_INHERIT
- PRIO_INHERIT
- PRIO_PROTECT

These constants are defined in the file **<pthread.h>**.

If an error occurs, **pthread_mutexattr_getprotocol()** returns an error number.

Errors

[EINVAL]	The object pointed to by <i>attr</i> does not exist.
[EPERM]	The application program is not sufficiently privileged.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.4.1
- **pthread_mutexattr_init()**, p. 581
- **pthread_mutexattr_setprotocol()**, p. 585

pthread_mutexattr_getpshared()

Synopsis

```
#include <pthread.h>

int pthread_mutexattr_getpshared( const pthread_mutexattr_t *attr,
                                  int *pshared );
```

Library

rtxLib.a

Description

pthread_mutexattr_getpshared() gets the **pshared** attribute from the mutex attributes object pointed to by *attr*.

The value of *pshared* is either **PTHREAD_PROCESS_SHARED**, which indicates that a mutex is owned by the kernel thread group, or **PTHREAD_PROCESS_PRIVATE**, which indicates that a mutex is owned by the creating thread group and will be destroyed when the creating thread group is destroyed.

The mutex attributes object must be initialized by **pthread_mutexattr_init()** or unpredictable results may occur.

If successful, **pthread_mutexattr_getpshared()** returns 0. Otherwise, **pthread_mutexattr_getpshared()** returns an error number.

Errors

[EINVAL] The object pointed to by *attr* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **pthread_mutexattr_destroy()**, p. 577
- **pthread_mutexattr_init()**, p. 581
- **pthread_mutexattr_setpshared()**, p. 586
- **pthread_tgdestroy_np()**, p. 599

Synopsis

```
#include <pthread.h>
int pthread_mutexattr_init( pthread_mutexattr_t *attr );
```

Library

rtxLib.a

Description

pthread_mutexattr_init() creates and initializes a mutex attributes object and stores its identifier in the field pointed to by *attr*.

Each attribute in the newly created object is set to its system default value.

pthread_mutexattr_init() always returns 0.

Errors

None.

Example

The following example provides several functions that manage the mutually exclusive use of arrays of control blocks. Each array of control blocks is guarded by a mutex. In this example the definition of the control is not important.

```
#include <pthread.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>

#define Block 0
#define NoBlock 1

/* The following definition of the control_block structure is */
/* provided for compiling the example only. Its true contents */
/* are not important for this example */
struct control_block;
{
    int dummy;
}

typedef struct locking_block
{
    struct locking_block *next; /* next LB in the list */
    pthread_mutex_t lock;      /* mutex guarding this array */
    pthread_t thread_id;        /* pthread currently holding mutex */
    int num_elems;              /* # of elements in the array */
    int current_elem;           /* # of first free control block */
}
```

pthread_mutexattr_init()

```
    struct control_block *start_addr; /* starting address of array */
} LB;

LB *lock_list = NULL;

LB *add_lock(int no_elems)
{
    LB *wp;
    pthread_mutexattr_t lock_mutex_attr;
    /* allocate storage for new locking_block */
    wp = (LB *)malloc(sizeof(LB));
    memset(wp, '\0', sizeof(LB));
    /* Create mutex attribute, and set protocol to priority protected */
    /* and priority ceiling to 10 */
    pthread_mutexattr_init(&lock_mutex_attr);
    pthread_mutexattr_setprotocol(&lock_mutex_attr, PRIO_PROTECT);
    pthread_mutexattr_setprioceiling(&lock_mutex_attr, 10);
    /* Create the array's mutex */
    pthread_mutex_init(&wp->lock, &lock_mutex_attr);
    /* Destroy mutex attribute because it is no longer needed */
    pthread_mutexattr_destroy(&lock_mutex_attr);
    /* Initialize remaining fields in the LB */
    wp->num_elems = no_elems;
    /* Allocate storage for array of control blocks */
    wp->start_addr = (struct control_block *)calloc(no_elems,
                                                    sizeof(struct control_block));
    /* add new LB to the lined list */
    wp->next = lock_list;
    lock_list = wp;
    return(wp);
}

int del_lock(LB *old)
{
    LB *wp;
    /* If the LB's mutex is currently locked */
    if(old->thread_id != 0)
        return(-1);
    /* Free the array of control blocks */
    free(old->start_addr);
    /* destroy the array's mutex */
    pthread_mutex_destroy(&old->lock);
    /* remove LB from the linked list */
    if(old == lock_list)
        lock_list = old->next;
    else
    {
```

```
        for(wp = lock_list; wp->next != old; wp = wp->next);
        wp->next = old->next;
    }
    /* free storage for LB */
    free(old);
    return(0);
}

pthread_t lock_control(LB *l, int flag)
{
    /* Do not block waiting for mutex */
    if(flag == NoBlock)
    {
        /* Acquire mutex */
        if(pthread_mutex_trylock(&l->lock) == 0)
            l->thread_id = pthread_self();
    }
    /* Block waiting for mutex */
    else
    {
        pthread_mutex_lock(&l->lock);
        l->thread_id = pthread_self();
    }
    return(l->thread_id);
}

int unlock_control(LB *l)
{
    /* if current thread does not hold the mutex */
    if(l->thread_id != pthread_self())
        return(-1);
    pthread_mutex_unlock(&l->lock);
    l->thread_id = 0;
    return(0);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §11.3.1
- **pthread_mutex_init()**, p. 571
- **pthread_mutexattr_destroy()**, p. 577

pthread_mutexattr_setprioceiling()

Synopsis

```
#include <pthread.h>

int pthread_mutexattr_setprioceiling( pthread_mutexattr_t *attr,
                                     int prioceiling );
```

Library

rtxLib.a

Description

pthread_mutexattr_setprioceiling() sets the **prioceiling** attribute in the mutex attributes object specified by *attr*.

attr must be initialized using **pthread_mutexattr_init()** or unpredictable results may occur.

prioceiling must be an integer in the inclusive range of the constants **PTHREAD_PRIO_MIN_NP** through **PTHREAD_PRIO_MAX_NP**. These constants are defined in the file **<pthread.h>**.

If successful, **pthread_mutexattr_setprioceiling()** returns 0. Otherwise, **pthread_mutexattr_setprioceiling()** returns an error number.

Errors

[EINVAL]	The object pointed to by <i>attr</i> does not exist or the value specified by <i>prioceiling</i> is not valid.
[EPERM]	The application program is not sufficiently privileged.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.4.1
- **pthread_mutexattr_getprioceiling()**, p. 578
- **pthread_mutexattr_init()**, p. 581

Synopsis

```
#include <pthread.h>

int pthread_mutexattr_setprotocol( pthread_mutexattr_t *attr,
                                   int protocol );
```

Library

rtxLib.a

Description

pthread_mutexattr_setprotocol() sets the **protocol** attribute in the mutex attributes object pointed to by *attr*.

attr must be initialized using **pthread_mutexattr_init()** or unpredictable results may occur. The value specified by *protocol* must be one of the following:

- NO_PRIO_INHERIT
- PRIO_INHERIT
- PRIO_PROTECT

If successful, **pthread_mutexattr_setprotocol()** returns 0. Otherwise, **pthread_mutexattr_setprotocol()** returns an error number.

Errors

[EINVAL]	The object pointed to by <i>attr</i> does not exist.
[ENOSUP]	The protocol specified by <i>protocol</i> is not valid.
[EPERM]	The application program is not sufficiently privileged.

Example

See **pthread_mutexattr_init()**, p. 581.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.4.1
- **pthread_mutexattr_getprotocol()**, p. 579
- **pthread_mutexattr_init()**, p. 581

pthread_mutexattr_setpshared()

Synopsis

```
#include <pthread.h>

int pthread_mutexattr_setpshared( pthread_mutexattr_t *attr,
                                  int pshared );
```

Library

rtxLib.a

Description

pthread_mutexattr_setpshared() sets the process shared attribute to *pshared* in the mutex attribute object pointed to by *attr*.

The value of *pshared* must be either **PTHREAD_PROCESS_SHARED**, which specifies that the new mutex will be owned by the kernel thread group, or **PTHREAD_PROCESS_PRIVATE**, which specifies that the new mutex will be owned by the creating thread group and will be destroyed when the creating thread group is destroyed.

The mutex attributes object must be initialized by **pthread_mutexattr_init()** or unpredictable results may occur.

If successful, **pthread_mutexattr_setpshared()** returns 0. Otherwise, **pthread_mutexattr_setpshared()** returns an error number.

Errors

[EINVAL]	The value of <i>pshared</i> is not valid or the object pointed to by <i>attr</i> does not exist.
----------	--

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **pthread_mutexattr_destroy()**, p. 577
- **pthread_mutexattr_init()**, p. 581
- **pthread_mutexattr_getpshared()**, p. 580
- **pthread_tgdestroy_np()**, p. 599

Synopsis

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once( pthread_once_t *once_control,
                 void (*init_routine)(void) );
```

Library

rtxLib.a

Description

pthread_once(), when first called by any thread with an object that points to *once_control*, calls *init_routine* with no arguments.

Subsequent calls to **pthread_once()** do not call *init_routine*.

init_routine is guaranteed to have completed when **pthread_once()** returns. *once_control* determines whether the associated initialization routine was called.

The application declares a single *once_control* pointer to each initialization routine and always passes the *once_control* pointer and *init_routine* to **pthread_once()**.

Results are undefined if *once_control* has storage class of automatic or is not initialized.

If successful, **pthread_once()** returns 0. Otherwise, **pthread_once()** returns an error number.

Errors

None.

Example

See **mempool_init()**, p. 420.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §16.1.8

pthread_resume_np()

Synopsis

```
#include <pthread.h>
int pthread_resume_np( pthread_t *thread );
```

Library

rtxLib.a

Description

pthread_resume_np() resumes the thread given by *thread*. Only a thread suspended using **pthread_suspend_np()** can be resumed using this call.

If successful, **pthread_resume_np()** returns 0. Otherwise, **pthread_resume_np()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **pthread_suspend_np()**, p. 596

Synopsis

```
#include <pthread.h>
pthread_t pthread_self( void );
```

Library

rtxLib.a

Description

pthread_self() returns the thread ID of the calling thread.

Errors

None.

Example

See **pthread_cond_init()**, p. 540.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §16.1.6

pthread_setcancelstate()

Synopsis

```
#include <pthread.h>

int pthread_setcancelstate( int state,
                           int *oldstate );
```

Library

rtxLib.a

Description

pthread_setcancelstate() sets the calling thread's cancelability state to *state* and returns the previous cancelability state in the location pointed to by *oldstate*.

The value of *state* must be one of the following constants, which are defined in **<pthread.h>**:

- PTHREAD_CANCEL_ENABLE
- PTHREAD_CANCEL_DISABLE

If **pthread_setcancelstate()** is called with **PTHREAD_CANCEL_ENABLE**, a cancellation point occurs in the calling thread when the state is set.

If successful, **pthread_setcancelstate()** returns 0. Otherwise, **pthread_setcancelstate()** returns an error number.

Errors

[EINVAL] *state* is not valid.

Example

See **fs_init()**, p. 231.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §18.2.2
- **pthread_setcanceltype()**, p. 591
- **pthread_testcancel()**, p. 597

Synopsis

```
#include <pthread.h>

int pthread_setcanceltype( int type,
                           int *oldtype );
```

Library

rtxLib.a

Description

pthread_setcanceltype() sets the calling thread's cancelability type to the value specified by *type* and returns the previous cancelability type in the location pointed to by *oldtype*.

The value specified by *type* must be one of the following:

- PTHREAD_CANCEL_DEFERRED
- PTHREAD_CANCEL_ASYNCHRONOUS

These constants are defined in **<pthread.h>**.

If successful, **pthread_setcanceltype()** returns 0. Otherwise, **pthread_setcanceltype()** returns an error number.

Errors

[EINVAL] The value specified by *type* is not valid.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §18.2.2
- **pthread_setcancelstate()**, p. 590
- **pthread_testcancel()**, p. 597

pthread_setschedparam()

Synopsis

```
#include <pthread.h>
#include <sched.h>

int pthread_setschedparam( pthread_t thread,
                           int *policy,
                           const struct sched_param *param );
```

Library

rtxLib.a

Description

pthread_setschedparam() sets the scheduling policy and associated scheduling parameters for the thread pointed to by *thread*, using values provided in *policy* and *param*, respectively.

For **SCHED_FIFO** and **SCHED_RR**, the valid values for the OS Open implementation, the only scheduling attribute is priority.

The value of *param->sched_priority* must be an integer value in the inclusive range of **PTHREAD_PRIO_MIN_NP** through **PTHREAD_PRIO_MAX_NP**. These constants are defined in the file **<pthread.h>**.

If successful, **pthread_setschedparam()** returns 0. Otherwise, **pthread_setschedparam()** returns an error number.

Errors

[EINVAL]	The value of <i>policy</i> is not valid or the value of <i>param->sched_priority</i> is not in the proper range.
[EPERM]	The application program is not sufficiently privileged.
[ESRCH]	The thread pointed to by <i>thread</i> does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §13.3.2
- **pthread_attr_init()**, p. 519
- **pthread_create()**, p. 552
- **pthread_getschedparam()**, p. 560

Synopsis

```
#include <pthread.h>

int pthread_setspecific( pthread_key_t key,
                        const void *value );
```

Library

rtxLib.a

Description

pthread_setspecific() 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 effect of calling this function either explicitly or implicitly from a thread specific data destructor function is undefined.

If successful, **pthread_setspecific()** returns 0. Otherwise, **pthread_setspecific()** returns an error number.

Errors

[EINVAL]	The value of <i>key</i> is not valid.
[ENOMEM]	Insufficient memory.

Example

See **pthread_key_create()**, p. 564.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §17.1.2
- **pthread_getspecific()**, p. 561
- **pthread_key_create()**, p. 564

pthread_sigmask()

Synopsis

```
#include <signal.h>

int pthread_sigmask( int how, const sigset_t *set, sigset_t *oset );
```

Library

rtxLib.a

Description

pthread_sigmask() examines or changes (or both) the signal mask of the calling thread.

If *set* is not NULL, it points to a signal set that changes the current signal mask in a manner controlled by *how*. The signal set must have been initialized by a signal set handling function (**sigaddset()**, **sigdelset()**, **sigemptyset()**, or **sigfillset()**).

how can have the following control values, which are defined in the file **<signal.h>**:

SIG_BLOCK	The resulting set is the union of the current signal mask and the signal set pointed to by <i>set</i> .
SIG_UNBLOCK	The resulting set is the intersection of the current signal mask and the complement of the signal set pointed to by <i>set</i> .
SIG_SETMASK	The resulting signal mask is the signal set pointed to by <i>set</i> .

If *oset* is not NULL, the previous mask is stored in the signal set object pointed to by *oset*. If *set* is NULL, it and *how* are ignored. This usage returns the current mask unchanged.

If successful, **pthread_sigmask()** returns 0. Otherwise, **pthread_sigmask()** returns an error number.

Errors

[EINVAL] *how* is not a defined value.

Example

The following code fragment blocks all signals and then restores the signal mask to its original value.

```
•
•
sigset_t lset;      /* signal set to block all signals */
sigset_t save_set; /* signal set to receive current mask */

sigfillset(&lset); /* create mask with all signals included */
pthread_sigmask(SIG_BLOCK, &lset, &save_set);
/* all signals now blocked */

/* finished with special section, now restore signal mask */
pthread_sigmask(SIG_SETMASK, &save_set, NULL);
•
•
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.4a/D8, §3.3.5
- **sigaddset()**, p. 740
- **sigdelset()**, p. 741
- **sigemptyset()**, p. 742
- **sigfillset()**, p. 743

pthread_suspend_np()

Synopsis

```
#include <pthread.h>
int pthread_suspend_np( pthread_t *thread );
```

Library

rtxLib.a

Description

pthread_suspend_np() suspends the thread given by *thread*. Only a ready or running thread can be suspended.

If successful, **pthread_suspend_np()** returns 0. Otherwise, **pthread_suspend_np()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **pthread_resume_np()**, p. 588

Synopsis

```
#include <pthread.h>
void pthread_testcancel( void );
```

Library

rtxLib.a

Description

pthread_testcancel() creates a cancelation point in the calling thread.

pthread_testcancel() has no effect if cancelability is disabled.

pthread_testcancel() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §18.2.2
- **pthread_setcancelstate()**, p. 590
- **pthread_setcanceltype()**, p. 591

pthread_tgcreate_np()

Synopsis

```
#include <pthread.h>
int pthread_tgcreate_np( tg_t *thread_group);
```

Library

rtx.o

Description

pthread_tgcreate_np() creates a new thread group. The new thread group ID is returned in the variable pointed to by *thread_group*.

A maximum of **PTHREAD_MAXTG_NP** thread groups can exist in the system at one time.

If successful, **pthread_tgcreate_np()** returns 0. Otherwise, **pthread_tgcreate_np()** returns an error number.

Errors

[ENOMEM]	Insufficient memory; the maximum number of thread groups already exist in the system.
----------	---

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

References

- **pthread_tgdestroy_np()**, p. 599

Synopsis

```
#include <pthread.h>
int pthread_tgdestroy_np( tg_t thread_group);
```

Library

rtx.o

Description

pthread_tgdestroy_np() destroys the thread group pointed to by *thread_group*.

All threads, mutexes, condition variables, timers, local semaphores, and message queue descriptors belonging to the thread group are also destroyed, but the message queue is not destroyed, even if **mq_unlink()** was called previously by another thread. Memory allocated for *thread_group* is freed. Thread group destructor functions are called at least **PTHREAD_DESTRUCTOR_ITERATIONS** times, in case a destructor function creates more thread group-specific data. Individual thread destructor functions are not executed.

The **PTHREAD_KERNEL_NP** thread group can not be destroyed.

If successful, **pthread_tgdestroy_np()** returns 0. Otherwise, **pthread_tgdestroy_np()** returns an error number.

Example

The following example destroys the current thread group;

```
#include <pthread.h>
void destroy_self()
{
    pthread_tgdestroy_np(pthread_tgself_np());
}
```

Errors

[EINVAL]	The thread group <i>thread_group</i> does not exist or <i>thread_group</i> specifies a protected thread group.
----------	--

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

References

- **pthread_tgcreate_np()**, p. 598

pthread_tggetkernel_np()

Synopsis

```
#include <pthread.h>
tg_t pthread_tggetkernel_np( void );
```

Library

rtx.o

Description

pthread_tggetkernel_np() returns the thread group ID of the **PTHREAD_KERNEL_NP** thread group.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

References

- **pthread_tgcreate_np()**, p. 598
- **pthread_tgdestroy_np()**, p. 599
- **pthread_tgself_np()**, p. 606

Synopsis

```
#include <pthread.h>

void *pthread_tggetspecific_np( tgkey_t key );
```

Library

rtx.o

Description

pthread_tggetspecific_np() gets the value currently bound to the specified *key* and the current thread group.

If successful, **pthread_tggetspecific_np()** returns the thread group-specific data value. If no thread group-specific data value is associated with *key* and *thread_group*, **pthread_tggetspecific_np()** returns NULL.

Errors

None.

Example

In the following example, *my_test()* gets the value currently bound to the external key variable *my_key*. If the value is NULL, *my_test()* adds a real page and sets the value bound to the key *my_key* to the address of the new page.

```
#include <pthread.h>
#include <vm.h>

void my_test()
{
    extern tgkey my_key;
    main_page= thread_tggetspecific_np(my_key);
    if (main_page == NULL)
    {
        vm_allocate(pthread_tgself_np(), &main_page, NULL, PAGE_SIZE, 1,
            VM_PROTECT_ALL);
        vm_addrealpages((pthread_tgself_np(), main_page, PAGE_SIZE, NULL, 0);
        pthread_tgsetspecific(my_key, main_page);
    }
    return;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

pthread_tggetspecific_np()

References

- [pthread_tgkeycreate_np\(\)](#), p. 603
- [pthread_tgkeydelete_np\(\)](#), p. 605
- [pthread_tgsetspecific_np\(\)](#), p. 607

Synopsis

```
#include <pthread.h>

int pthread_tgkeycreate_np( tgkey_t *key
                           void (*destructor(void *)));
```

Library

rtx.o

Description

pthread_tgkeycreate_np() creates a key, visible to all thread groups, and stores the key value in an object pointed to by *key*.

When a key is created, the value NULL is associated with the new key in all active thread groups. When a thread group is created, the value NULL is associated with all defined keys in the new thread group.

An optional destructor function can be associated with each key. When a thread group is destroyed, and if a key has a non-NULL destructor pointer and a thread group associates a non-NULL value with that key, the destructor function is called with the current key value as its sole argument. The order of destructor calls is not specified when more than one destructor exists for a thread group.

Destructor calls are repeated as needed (at least **PTHREAD_DESTRUCTOR_ITERATIONS** times) in case a destructor function creates more thread group-specific data.

Thread executing this call must be in the **PTHREAD_PRIVILEGED_NP** state.

If successful, **pthread_tgkeycreate_np()** returns 0. Otherwise, **pthread_tgkeycreate_np()** returns an error number.

Errors

[EAGAIN]	The number of keys created equals the maximum numbers of keys allowed.
----------	--

Example

In the following example, *my_init()* test to make sure it is in the kernel thread group. It then creates a key and places its value in the global *my_key*.

```
#include <vm.h>
#include <pthread.h>

tgkey_t my_key;
int my_init()
{
    int rc;
    if (pthread_tgself_np() != pthread_tggetkernel_np())
```

pthread_tgkeycreate_np()

```
    return(-1);
rc = pthread_tgkeycreate_np(&my_key,NULL);
if(rc!=0)
    return (-1);
return 0;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	No

References

- **pthread_tggetspecific_np()**, p. 601
- **pthread_tgsetspecific_np()**, p. 607

Synopsis

```
#include <pthread.h>
int pthread_tgkeydelete_np( tgkey_t key);
```

Library

rtx.o

Description

pthread_tgkeydelete_np() deletes the thread group-specific data key named by *key*.

If thread group-specific data other than a NULL is associated with *key* when **pthread_tgkeydelete_np()** is called, the effect is undefined. Attempts to use *key* after **pthread_tgkeydelete_np()** is called results in undefined behavior.

Thread executing this call must be in the **PTHREAD_PRIVILEGED_NP** state.

If successful, **pthread_tgkeydelete_np()** returns 0. Otherwise, **pthread_tgkeydelete_np()** returns an error number.

Errors

[EINVAL] *key* is invalid.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	No

References

- **pthread_tgkeycreate_np()**, p. 603
- **pthread_tggetspecific_np()**, p. 601
- **pthread_tgsetspecific_np()**, p. 607

pthread_tgself_np()

Synopsis

```
#include <pthread.h>
tg_t pthread_tgself_np( void );
```

Library

rtx.o

Description

pthread_tgself_np() returns the thread group ID of the calling thread.

Errors

None.

Example

Refer to **vm_addrealpages()**, p. 932.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

References

- **pthread_tgcreate_np()**, p. 598
- **pthread_tgdestroy_np()**, p. 599

Synopsis

```
#include <pthread.h>

int pthread_tgsetspecific_np( tgkey_t key, void *value );
```

Library

rtx.o

Description

pthread_tgsetspecific_np() associates the thread group-specific *value* with the current thread group and the key specified by *key*.

Different thread groups can bind different values to the same key.

If successful, **pthread_tgsetspecific_np()** returns 0. Otherwise, **pthread_tgsetspecific_np()** returns an error number.

Errors

[EINVAL]	The thread group <i>thread_group</i> does not exist or the value of <i>key</i> is not valid.
----------	--

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

References

- **pthread_tggetspecific_np()**, p. 601
- **pthread_tgkeycreate_np()**, p. 603
- **pthread_tgkeydelete_np()**, p. 605

pthread_whattg_np()

Synopsis

```
#include <pthread.h>
tg_t pthread_whattg_np( pthread_t thread );
```

Library

rtx.o

Description

pthread_whattg_np() returns the thread group ID of the thread specified by *thread*.

If successful, **pthread_whattg_np()** returns the thread group ID. Otherwise, **pthread_whattg_np()** returns 0, indicating *thread* refers to an invalid thread identifier.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

Synopsis

```
#include <stdio.h>

int putc( int c,
          FILE *stream );
```

Library

None. This is a macro defined in **<stdio.h>**.

Description

putc() converts *c* to an unsigned char and then writes *c* to the current position of the output stream pointed to by *stream*.

putc() returns the character written.

putc() and **fputc()** are similar, but **putc()** is a macro and **fputc()** is a function. The *stream* argument to **putc()** should not be an expression with side effects.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

The following example prints a string to a file one byte at a time.

```
#include <stdio.h>

int print_line(char *line, FILE *file)
{
    char *wp;
    wp = line;
    while(*wp != '\0')
    {
        /* if an error occurred while printing character */
        if(putc(*wp, file) == EOF)
        {
            perror("Write error");
            clearerr(file);
            return(EOF);
        }
        wp++;
    }
    return(0);
}
```

putc()

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.7.8
- **fputc()**, p. 222
- **fs_init()**, p. 231
- **getc()**, p. 255
- **getchar()**, p. 257

Synopsis

```
#include <stdio.h>
int putc_unlocked( int c,
                  FILE *file );
```

Library

fsLib.a

Description

putc_unlocked() is identical to **putc()** except that it is not async safe.

putc_unlocked() can be safely used only within code protected by **flockfile()** and **funlockfile()**.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **flockfile()**, p. 205.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §19.2.2.2
- **fs_init()**, p. 231

putchar()

Synopsis

```
#include <stdio.h>
int putchar( int c );
```

Library

None. This is a macro defined in **<stdio.h>**.

Description

putchar() converts *c* to an unsigned char and then writes *c* to the stdout stream.

putchar() returns the character written. A return value of EOF indicates an error.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **nanosleep()**, p. 466.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.7.9
- **fputc()**, p. 222
- **fs_init()**, p. 231
- **getc()**, p. 255
- **getchar()**, p. 257

Synopsis

```
#include <stdio.h>
int putchar_unlocked( int c );
```

Library

None. This is a macro defined in **<stdio.h>**.

Description

putchar_unlocked() is identical to **putchar()** except that it is not async safe.

putchar_unlocked() can be safely used only within code protected by **flockfile()** and **funlockfile()**.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

The following example prints a name to the standard output.

```
#include <stdio.h>

int prt_name(char *buffer)
{
    char *wp;
    int rv;
    /* lock stdout for exclusive use */
    flockfile(stdout);
    /* print the name to stdout */
    wp = buffer;
    while(*wp != '\0')
    {
        rv = putchar_unlocked(*wp++);
        if(rv == EOF);
        break;
    }
    /* Unlock stdout */
    funlockfile(stdout);
    return(rv);
}
```

putchar_unlocked()

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §19.2.2.2
- **fs_init()**, p. 231

Synopsis

```
#include <stdio.h>
int puts( const char *string );
```

Library

fsLib.a

Description

puts() writes the string pointed to by *string* to the standard output and appends a new line to the output.

The terminating NULL character is not written.

puts() returns EOF if an error occurs.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **strtok()**, p. 808.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.7.10
- **fputs()**, p. 224
- **fs_init()**, p. 231
- **gets()**, p. 289

qsort()

Synopsis

```
#include <stdlib.h>

void qsort( void *base,
            size_t num,
            size_t width,
            int(*compare)(const void *element1, const void *element2) );
```

Library

cLib.a

Description

qsort() sorts an array of *num* elements, each containing *width* bytes.

base is a pointer to the base of the array to be sorted. **qsort()** writes over the array with the sorted elements.

compare points to a programmer-supplied routine that compares two array elements and returns a value specifying their relationship. **qsort()** calls this routine one or more times during the sort, passing pointers to two array elements on each call. The routine must compare the elements and then return a value.

Less than 0	<i>element1</i> is less than <i>element2</i>
0	<i>element1</i> is identical to <i>element2</i>
Greater than 0	<i>element1</i> is greater than <i>element2</i>

The sorted array elements are stored in the order defined by the comparison routine. Reversing the sense of “greater than” and “less than” in the comparison routine reverses the sorting order. The order of elements is unspecified when two elements compare equally.

qsort() returns nothing.

Errors

None.

Example

See **bsearch()**, p. 86.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.5.2
- **bsearch()**, p. 86

Synopsis

```
#include <queLib.h>

void queCreate( que_t *que, void *startaddr,
               unsigned long number_of_nodes,
               unsigned long node_size );
```

Library

queLib.a

Description

queCreate() creates *number_of_nodes* queue data type (**queNode_t**) of *node_size* size, in bytes starting at the address *startaddr*, and adds the queue data type to the root queue data type pointed to by *que*. *que_t* must be initialized prior to calling **queCreate()**.

queCreate() returns nothing.

Errors

None.

Example

```
/*
** TEST QUEUE SAMPLE
*/

typedef unsigned long ULONG;
typedef que_t TEST_QUEUE_S;
typedef struct test_queue_node_s
{
    queNode_t dblink;
    ULONG eye_catcher;
    ULONG element1;
    ULONG element2;
}TEST_QUEUE_NODE_S;
TEST_QUEUE_S test_q;      /* the root queue */
TEST_QUEUE_NODE_S *test_q_node_pool_p; /* ptr to memory for nodes */
void test_queue()
{
    int num_nodes = 3;
    int i;
```

queCreate()

```
TEST_QUEUE_NODE_S *node_p;
/*
** Allocate nodes for queueing
*/
test_q_node_pool_p =
    (TEST_QUEUE_NODE_S *) malloc(sizeof(TEST_QUEUE_NODE_S) *
num_nodes);
/*
** initialize the fields of the structure
*/
node_p = test_q_node_pool_p;
for (i = 0;
    i < num_nodes;
    i++)
{
    node_p->eye_catcher = 0xFEEFACE;
    node_p->element1 = i * 0x11111111;
    node_p->element2 = i * 0x11111111;
    node_p++;
}/* endfor */
queInit(&test_q);
queCreate(&test_q, test_q_node_pool_p,
    num_nodes, sizeof(TEST_QUEUE_NODE_S) - sizeof(queNode_t));
return;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <queLib.h>
queNode_t *queDeq( que_t *que );
```

Library

queLib.a

Description

queDeq() removes the oldest node from the queue pointed to by *que* and returns the address of the removed node.

If the queue is empty, **queDeq()** returns a NULL pointer.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

queDeqNoCheck()

Synopsis

```
#include <queLib.h>
queNode_t *queDeqNoCheck( que_t *que );
```

Library

queLib.a

Description

queDeqNoCheck() removes the oldest node from the queue pointed to by *que* and returns the address of the removed node.

Results of **queDeqNoCheck()** are undefined if the queue is empty, but **queDeqNoCheck()** does not check whether the queue is empty.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <queLib.h>
void queEnq( que_t *que, const queNode_t *node );
```

Library

queLib.a

Description

queEnq() places the node pointed to by *node* at the end of the queue pointed to by *que*.

queEnq() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

queEnqFront()

Synopsis

```
#include <queLib.h>
void queEnqFront( que_t *que, const queNode_t *node );
```

Library

queLib.a

Description

queEnqFront() places the node pointed to by *node* at the front of the queue pointed to by *que*.

queEnqFront() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <queLib.h>
void queInit( que_t *que );
```

Library

queLib.a

Description

queInit() initializes the root queue data type, pointed to by *que*, in a doubly linked list of queue nodes.

queInit() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

queInsert()

Synopsis

```
#include <queLib.h>
void queInsert( queNode_t *node, queNode_t *new );
```

Library

queLib.a

Description

queInsert() inserts the node pointed to by *new* in the queue after the node pointed to by *node*.

queInsert() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <queLib.h>

queNode_t *queNewer( que_t *que, const queNode_t *node );
```

Library

queLib.a

Description

queNewer() returns the address of the next node in the queue pointed to by *que* that is newer than the node pointed to by *node*.

queNewer() returns a NULL pointer if the node pointed to by *node* is the newest node in the queue.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

queNewest()

Synopsis

```
#include <queLib.h>
queNode_t *queNewest( que_t *que );
```

Library

queLib.a

Description

queNewest() returns the address of the newest node in the queue pointed to by *que*.

queNewest() returns a NULL pointer if the queue is empty.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <queLib.h>

queNode_t *queOlder( que_t *que, const queNode_t *node );
```

Library

queLib.a

Description

queOlder() returns the address of the next node, in the queue pointed to by *que*, that is older than the node pointed to by *node*.

queOlder() returns a NULL pointer if the node pointed to by *node* is the oldest node in the queue.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

queOldest()

Synopsis

```
#include <queLib.h>
queNode_t *queOldest( que_t *que );
```

Library

queLib.a

Description

queOldest() returns the address of the oldest node in the queue pointed to by *que*.

queOldest() returns a NULL pointer if the queue is empty.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <queLib.h>

void queRemove( queNode_t *node );
```

Library

queLib.a

Description

queRemove() removes the node pointed to by *node* from the queue that is linked with the node.

queRemove() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

raise()

Synopsis

```
#include <signal.h>
int raise( int sig );
```

Library

cLib.a

Description

raise() sends the signal named by *sig* to the calling thread.

Calling **raise()** is equivalent to calling:

```
pthread_kill(pthread_self(), sig)
```

sig is not consumed by the interrupted signal call. A signal remains pending until consumed by **sigwait()**.

If *sig* is 0, error checking is performed, but no signal is sent.

If successful, **raise()** returns 0. Otherwise, **raise()** returns -1 and sets *errno*.

Errors

[EINVAL] *sig* is not a supported signal number.

Example

See **getchar_unlocked()**, p. 258.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.7.2.1
- **kill()**, p. 350
- **pthread_kill()**, p. 568

Synopsis

```
#include <stdlib.h>
int rand( void );
```

Library

cLib.a

Description

rand() selects a pseudo-random integer in the range 0 through **RAND_MAX** (a macro defined in **<stdlib.h>**).

srand() is typically called before calling **rand()** to set the seed for the random number generator. At first invocation, the default seed is 1.

Errors

None.

Example

See **fread()**, p. 225.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.2.1
- **rand_r()**, p. 632
- **srand()**, p. 765

rand_r()

Synopsis

```
#include <stdlib.h>

int rand_r( unsigned int *seed );
```

Library

cLib.a

Description

rand_r() is a reentrant version of **rand()**.

rand_r() returns a pseudo-random number as described by **rand()**. *seed* is used to generate the pseudo-random number and store the next seed.

Errors

None.

Example

See **bsearch()**, p. 86.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Std 1003.4a/D8, §19.2.5.3
- **rand()**, p. 631

Synopsis

```
#include <unistd.h>

ssize_t read( int filedes, void *buffer, size_t nbytes );
```

Library

devLib.a

Description

read() tries to read *nbytes* bytes from the file associated with the open file descriptor *filedes* into the buffer pointed to by *buffer*.

If the file can seek, **read()** starts from and updates the file offset.

read() returns the number of bytes read and placed into the buffer pointed to by *buffer*. This number can be less than *nbytes* if the file is smaller than *nbytes*, a blocking **read()** was interrupted by a signal, or a special file has less than *nbytes* bytes immediately available for reading.

If *nbytes* is 0, **read()** returns 0 with no other results. When **read()** tries to read from a device file that supports non-blocking reads and there no data is available, **read()** returns -1 and sets *errno* to **[EAGAIN]** if **open()** specified **O_NONBLOCK**. Otherwise, **read()** blocks until data becomes available.

If successful, **read()** returns the number of bytes transferred to the buffer. Otherwise, **read()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>filedes</i> is not a valid file descriptor.
[EAGAIN]	Non-blocking reads are supported, O_NONBLOCK was set for <i>filedes</i> , and read() would block because of insufficient data.
[EINTR]	A unblocked signal is pending on the system or the thread.

Example

The following example reads and writes data to an asynchronous device driver.

```
#include <sys/devLib.h>
#include <sys/devDrive.h>
#include <sys/ioctl.h>
#include <types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdarg.h>
#include <stdio.h>
```

read()

```
#include <sys/asyncLib.h>
#include <config.h>
#include <iocclib.h>

int async_init(driver_t *dsw, va_list vargs);
void main();
void panic();
const conf_t _Kernel_Config_block = {
    main,          /* Main function */
    panic,         /* Panic function */
    0x00000000,    /* Main argc */
    0x00040000,    /* Main stack size */
    (void *)0x00000000, /* Memory heap 1 start address */
    0x00500000,    /* Memory heap 1 size */
    0x00000000,    /* Memory heap 2 start address */
    0x00000000,    /* Memory heap 2 size */
    0x000004b0,    /* Trace table size */
    0x00fe5100,    /* Tick rate */
    0x0fe51000     /* Round robin time slice */
};

void main()
{
    int devhandle, numbytes = 0;
    int fd1, fd2;
    char buffer[120];
    int rv;
    char *wp;

    /* Call Platform specific calls to initialize hardware */
    iocclnit();
    timertick_install();
    /* Initialize device driver subsystem */
    rv = dev_io_init(16, 16);
    if(rv < 0)
        abort();
    /* Identify the async device driver to the device subsystem */
    rv = driver_install(&devhandle, async_init);
    if(rv < 0)
        abort();
    /* Associate the async device driver with the name "dev/tty0" */
    rv = device_install("/dev/tty0", CHRTYPE, devhandle, 1, 128, 128);
    if(rv < 0)
        abort();
    /* Open to two file descriptors to the async driver */
    /* one for read only and one for write only */
}
```

```

fd1 = open("/dev/tty0", O_RDONLY, asyncParityNone, asyncParityOdd,
           asyncStopBits1, asyncDataBits8, 9600);
fd2 = open("/dev/tty0", O_WRONLY, asyncParityNone, asyncParityOdd,
           asyncStopBits1, asyncDataBits8, 9600);
if((fd1 < 0) || (fd2 < 0))
    abort();
/* Read from the async device one byte at a time until a */
/* new line character is entered */
wp = buffer;
while(read(fd1, wp, 1) == 1)
{
    if(*wp == '\n')
        break;
    numbytes++;
    wp++;
}

/* Write to the device the number of bytes read */
sprintf(buffer, "Number of bytes read: %d\n", numbytes);
write(fd2, (void *)buffer, strlen(buffer));

/* close the file descriptors */
close(fd1);
close(fd2);
}

```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §6.4.1
- **open()**, p. 478

readdir()

Synopsis

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

struct dirent *readdir( DIR *dirp );
```

Library

devLib.a

Description

readdir() returns a pointer to a structure representing the directory entry at the file position indicator in the directory stream pointed to by *dirp* and positions the directory stream at the next entry.

When **readdir()** encounters the end of the directory, it returns a NULL.

If an error occurs, **readdir()** returns a NULL and sets *errno*.

Errors

[EBADF] *dirp* does not point to an open directory stream.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.1.2.2

Synopsis

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

int readdir_r( DIR *dirp, struct dirent *entry, struct dirent **result );
```

Library

devLib.a

Description

readdir_r() stores a pointer to a structure representing the directory entry at the file position indicator in the directory stream and positions the directory stream at the next entry.

entry points to a directory entry in the application program's storage. If **readdir_r()** is successful, the pointer referenced by *result* is set to *entry*. If the end of the directory stream has been reached, the pointer is set to NULL.

If successful, **readdir_r()** returns 0. Otherwise, **readdir_r()** returns an error number.

Errors

[EBADF]	<i>dirp</i> does not point to an open directory stream.
[EIO]	A physical input/output (I/O) error occurred.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Std 1003.4a/D8, §19.2.4.2
- **readdir()**, p. 636

realloc()

Synopsis

```
#include <stdlib.h>

void *realloc( void *mem_address, size_t newsize );
```

Library

cLib.a

Description

realloc() changes the size of an allocated block of memory to a new size given by *newsize*.

The address of the block is specified by the pointer *mem_address*. The pointer must be NULL or a value returned by a previous call to **malloc()**, **calloc()**, or **realloc()**. If *mem_address* is NULL, **realloc()** behaves like **malloc()** and allocates a new block of memory of size *newsize*. If *newsize* is 0, **realloc()** behaves like **free()** and frees the memory pointed to by *mem_address*.

The return pointer of the altered block may be different, but, up to the lesser of the original and new size, the contents of the block are guaranteed to be unchanged.

If successful, **realloc()** returns a pointer to the allocated memory. Otherwise, the *mem_address* pointer is not changed and **realloc()** returns NULL.

Errors

None.

Example

The following example either allocates storage for an array of long with a given number of elements or changes the size of the storage for an already existing array.

```
#include <stdlib.h>

long *stor_mang(long *addr, int num_elems)
{
    /* if addr is NULL allocate storage for a new array */
    /* free existing storage if num_elems equals zero */
    addr = (long *)realloc(addr, num_elems * sizeof(long));
    return(addr);}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.3.4
- **calloc()**, p. 89
- **free()**, p. 228
- **malloc()**, p. 398

recv()

Synopsis

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

int recv( int s, void *msg, int len, int flags );
```

Library

netLib.a

Description

recv() allows an application program to receive messages through a connected socket named by *s*.

This call is equivalent to **recvfrom()**, except that the destination is not specified. *len* specifies the length of the message. *msg* points to the buffer that holds the message.

If the message cannot fit in the supplied buffer, excess bytes may be truncated, depending on the type of socket that issued the message. If no messages are available at the socket, **recv()** waits for a message to arrive unless the socket is non-blocking. If a socket is non-blocking, OS Open returns an error.

flags controls message reception. *flags* is formed by applying a logical OR to one or both of the following constants, which are defined in the file **<sys/socket.h>**:

MSG_PEEK	Peeks at incoming message. The data is treated as unread and the next recv() (or similar call) returns this data.
MSG_OOB	Process out-of-band data. The underlying protocol must support out-of-band data, for example, TCP.

If successful, **recv()** returns the number of characters received. Otherwise, **recv()** returns `-1` and sets *errno*.

Errors

[EBADF]	The socket descriptor <i>s</i> is not valid.
[ENOTSOCK]	The socket descriptor <i>s</i> references a file, not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking, and the receive operation would block.
[EINTR]	A signal interrupted recv() before any data was available.

Example

The following example receives a single character on a socket. The routine creates and names a socket and binds it to an address. The routine then waits for a socket connection. When the connection is established the routine starts a loop in which characters can be received.

```
#include <stdio.h>
#include <types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/select.h>

int ex_recv(void)
{
    int maxsock; /* Max # of sockets to select on */
    int *allsock, *readable; /* File descriptor sets */
    int fdsetsize; /* Number bytes in fd set */
    int lsd; /* New socket descriptor */
    struct sockaddr_in lsa; /* Used in accept() call */
    int ia = sizeof lsa.sin_addr; /* Used in accept() call */
    int n; /* # of sockets with data waiting */
    int sd;
    struct sockaddr_in sa;
    struct servent *service;
    int rc;
    char *recvchr = " ";
    bzero((char *)&sa, sizeof(sa));
    if (getservlock())
        return -1;
    sa.sin_port=50003;
    getservunlock();
    if ((sd=socket(AF_INET,SOCK_STREAM,0))<0) {
        (void)printf("socket() failed\n");
        return(-1);
    }
    sa.sin_addr.s_addr=INADDR_ANY;
    sa.sin_family=AF_INET;
    if (bind(sd, (struct sockaddr *)&sa, sizeof(struct sockaddr_in))<0) {
        (void)printf("bind() failed\n");
        return(-1);
    }
    if (listen(sd,5)<0) {
        printf("listen() failed\n");
        return(-1);
    }
    fdsetsize=32;
```

recv()

```
allsock=(int *)malloc(fdsetsize);
if (NULL==allsock)
    return((void *)-1);
readable=(int *)malloc(fdsetsize);
if (NULL==readable) {
    free(allsock);
    return((void *)-1);
}
memset(allsock,0,fdsetsize);
FD_SET(sd,allsock);          /* Add socket to list for select */
maxsock = sd+1;
memcpy(readable,allsock,fdsetsize);
printf("entering select, sd = %d\n", sd);
n = select(maxsock, readable, NULL, NULL, NULL);
if (0==n) {
    printf("select() returned 0\n");
}
printf("waiting on an accept\n");
if ( (lfd=accept(sd,(struct sockaddr *)&lsa,&ia)) < 0) {
    printf("accept() failed\n");
    return((void *)-1);
}
printf("received a connection\n");
for(;;) {
    rc=recv(lfd, recvchr, 1, 0);
    printf("recv call returned %d, character received was %c\n", rc, *recvchr);
}
return(rc);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.10
- **getsockname()**, p. 295
- **read()**, p. 633
- **recvfrom()**, p. 643
- **select()**, p. 691
- **send()**, p. 713
- **socket()**, p. 760
- **write()**, p. 960

Synopsis

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

int recvfrom( int s, void *buffer, int buflen, int flags,
              struct sockaddr *from, int *fromlen );
```

Library

tcplib.a

Description

recvfrom() receives a message from a connected or unconnected socket named by *s*.

To return the source address of the message, specify a non-NULL value for the *from* argument. *fromlen* is a value-result parameter, initialized to the size of the buffer that *from* points to. On return, **recvfrom()** modifies the value pointed to by *fromlen* to indicate the size of the stored address.

When successful, **recvfrom()** returns the length of the received message. If a message cannot fit in the buffer pointed to by *buffer*, excess bytes may be truncated, depending on the type of socket that issued the message.

If no messages are available at the socket, **recvfrom()** waits for a message to arrive, unless the socket is non-blocking. If the socket is non-blocking, the system returns an error.

flags controls message reception. *flags* is formed by applying a logical OR to one or both of the following constants, which are defined in the file **<sys/socket.h>**:

MSG_PEEK	Peeks at incoming message. The data is treated as unread and the next recvfrom() (or similar call) returns this data.
MSG_OOB	Process out-of-band data. The underlying protocol must support out-of-band data, for example, TCP.

If an error occurs, **recvfrom()** returns `-1` and sets *errno*.

Errors

[EBADF]	The socket descriptor <i>s</i> is not valid.
[ENOTSOCK]	The socket descriptor <i>s</i> references a file, not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking, and the receive operation would block.
[EINTR]	The receive was interrupted by delivery of a signal before any data was available.

recvfrom()

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.10
- **getsockopt()**, p. 296
- **read()**, p. 633
- **recvmsg()**, p. 645
- **select()**, p. 691
- **send()**, p. 713
- **socket()**, p. 760
- **write()**, p. 960

Synopsis

```
#include <types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>

int recvmsg( int s, struct msghdr *msg, int flags );
```

Library

tcpipLib.a

Description

recvmsg() receives a message from a connected or unconnected socket, named by *s*, using the **msghdr** message structure.

recvmsg() returns the length of the message. If a message cannot fit in the buffer pointed to by *msg*, excess bytes may be truncated, depending on the type of socket that issued the message.

If no messages are available at the socket, **recvmsg()** waits for a message to arrive. If the socket is non-blocking and no messages are available, **recvmsg()** fails.

recvmsg() uses a **msghdr** structure to decrease the number of directly supplied parameters. The **msghdr** structure is defined in the file **<sys/socket.h>**.

```
struct msghdr {
    caddr_t      msg_name;
    u_int        msg_namlen;
    struct iovec *msg_iov;
    u_int        msg_iov_len;
    caddr_t      msg_control;
    u_int        msg_controllen;
    int          msg_flags;
}
```

msg_name and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be a null pointer if no names are desired or required. *msg_iov* and *msg_iovlen* describe the scatter gather locations. *msg_control* of length *msg_controllen* is a buffer for other protocol-related messages or other miscellaneous data. *msg_flags* is set on return in a way that may include one or both of the values of *flags*.

The returned value **MSG_EOR** indicates end-of-record; **MSG_TRUNC** indicates that some trailing datagram data was discarded; **MSG_CTRUNC** indicates that some control data was discarded because of insufficient space. **MSG_OOB** indicates that expedited data was received (out of band data).

flags controls message reception. *flags* is formed by applying a logical OR to one or both of the following constants, which are defined in the file **<sys/socket.h>**:

recvmsg()

MSG_PEEK	Peeks at incoming message. The data is treated as unread and the next recvmsg() (or similar call) returns this data.
MSG_OOB	Process out-of-band data. The underlying protocol must support out-of-band data, for example, TCP.

If an error occurs, **recvmsg()** returns `-1` and sets *errno*.

Errors

[EBADF]	The socket descriptor <i>s</i> is not valid.
[ENOTSOCK]	The socket descriptor <i>s</i> references a file, not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking, and the receive operation would block.
[EINTR]	The receive was interrupted by delivery of a signal before any data was available for the receive.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.10
- **getsockopt()**, p. 296
- **read()**, p. 633
- **recvfrom()**, p. 643
- **select()**, p. 691
- **send()**, p. 713
- **sendmsg()**, p. 716
- **socket()**, p. 760
- **write()**, p. 960

Synopsis

```
#include <kadtLib.h>

unsigned long reg_get( int reg, pthread_t thread );
```

Library

kadtLib.a

Description

reg_get() returns the contents of the register *reg* for *thread*. Valid values for *reg* are defined in **<sys/reg.h>**.

The thread should be suspended at a breakpoint before **reg_get()** is called.

If *thread* or *reg* are not valid, **reg_get()** returns the value 0xdeadbeef.

Errors

None.

Example

See **dbbrkpt_next_inst()**, p. 157.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **reg_set()**, p. 648

reg_set()

Synopsis

```
#include <kadtLib.h>

int reg_set( int reg, unsigned long data, pthread_t thread );
```

Library

kadtLib.a

Description

reg_set() writes *data* into the register *reg* for *thread*.

Valid values for *reg* are defined in the file **<sys/reg.h>**.

The thread should be suspended at a breakpoint before **reg_set()** is called.

reg_set() returns -1 if *thread* or *reg* are not valid. Otherwise, **reg_set()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **reg_get()**, p. 647

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

int registerrpc( u_long prognum, u_long versnum, u_long procnum,
                 char *(*procname)(), xdrproc_t inproc, xdrproc_t outproc );
```

Library

rpcLib.a

Description

registerrpc() registers a procedure with the RPC service package.

If a request arrives that matches the values of the *prognum*, *versnum*, and *procnum* parameters, the procedure specified by *procname* is called using a pointer to its parameters; it subsequently returns a pointer to its static results.

Note: Remote procedures registered in this form are accessed using the UDP/IP transport protocol only.

prognum specifies the program number of the remote procedure; *versnum* specifies its version number.

procnum identifies the number of the called procedure; *procname* identifies its name.

inproc specifies the XDR routine that encodes remote procedure parameters. *outproc* specifies the XDR routine that decodes the remote procedure results.

If successful, **registerrpc()** returns 0. Otherwise, **registerrpc()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **callrpc()**, p. 90
- **rpc_thread_init()**, p. 676
- **svcudp_create()**, p. 839

rename()

Synopsis

```
#include <stdio.h>

int rename( const char *old, const char *new );
```

Library

devLib.a

Description

rename() changes the name of a file or directory from the name pointed to by *old* to the name pointed to by *new*.

For a file rename operation, an existing file named by the new name must not be a directory. An existing file specified by *new* is processed as if it was the target of an **unlink()** function.

For a directory rename operation, if *new* exists, it must be an empty directory, not a file.

The path name containing the name pointed to by *new* cannot contain any path prefix naming *old*.

If successful, **rename()** returns 0. Otherwise, **rename()** returns -1 and sets *errno*.

Errors

[EACCES]	Search permission is denied for a component of the path, or the requested operation requires writing with write permission is denied, or a directory containing <i>old</i> or <i>new</i> denies write permission.
[EBUSY]	For file system implementations that mark directories as busy, the directory is being used by another function.
[EEXIST]	<i>new</i> points to a directory that is not empty.
[EINVAL]	The <i>new</i> directory path name contains a prefix that names the <i>old</i> directory.
[EISDIR]	<i>new</i> points to a directory, and <i>old</i> points to a file.
[ENAMETOOLONG]	A pathname component length exceeds {NAME_MAX} or the length of the existing or new file names exceeds {PATH_MAX} .
[EMLINK]	<i>old</i> is a directory, and the link count of the parent directory of <i>new</i> would exceed {LINK_MAX} .
[ENOENT]	<i>old</i> points to a file or directory name that does not exist or points to an empty string.
[ENOSPC]	The directory that would contain <i>new</i> cannot be extended.

[ENOTDIR]	A component of either path prefix is not a directory, or <i>old</i> names a directory and <i>new</i> names an existing regular file.
[EROFS]	The directory is on a read-only file system.
[EXDEV]	<i>new</i> and <i>old</i> are on different file systems.

Example

See **access()**, p. 55.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.5.3

rewind()

Synopsis

```
#include <stdio.h>

void rewind( FILE *stream );
```

Library

fsLib.a

Description

rewind() repositions the file position indicator associated with the stream pointed to by *stream* at the beginning of the file and clears the error indicator for the stream.

A call to the **rewind()** function is equivalent to:

```
(void) fseek(stream, 0, SEEK_SET);
```

except that **rewind()** clears the error indicator.

rewind() returns nothing.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

See **isalnum()**, p. 333.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.9.5
- **fseek()**, p. 235
- **ftell()**, p. 239

Synopsis

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

void rewinddir( DIR *dp );
```

Library

devLib.a

Description

rewinddir() resets the position indicator of the directory stream to the first entry, reflecting the state of the corresponding directory as if the stream had been processed by **opendir()**.
rewinddir() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.1.2.2
- **opendir()**, p. 480

rmdir()

Synopsis

```
#include <unistd.h>

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

Library

devLib.a

Description

rmdir() removes a directory specified by the name pointed to by *path*.

The directory is removed only if it is empty. If the directory is open when **rmdir()** is called, directory removal does not occur until all references to the directory are closed.

If successful, **rmdir()** returns 0. Otherwise, **rmdir()** returns -1 and sets *errno*.

Errors

[EACCES]	Search permission is denied for a component of the path, or the requested operation requires writing while write permission is denied.
[EEXIST]	The path argument names a directory that is not empty.
[ENAMETOOLONG]	A pathname component length exceeds {NAME_MAX} or the length of <i>path</i> exceeds {PATH_MAX} .
[ENOENT]	<i>path</i> names a directory that does not exist or points to an empty string.
[ENOSPC]	The file system has insufficient space to contain the new directory or to extend the parent directory.
[ENOTDIR]	A component of <i>path</i> is not a directory.
[EROFS]	The directory is on a read-only file system.

Example

See **mkdir()**, p. 427.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.5.2
- **mkdir()**, p. 427

Synopsis

```
#include <rmsLib.h>
d_queue_t *rms_delq( void );
```

Library

rmsLib.a

Description

rms_delq() removes the oldest node in the delta queue used by the rate monotonic scheduler (RMS) thread.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **rms_init()**, p. 656
- **rms_insq()**, p. 657
- **rms_oldest()**, p. 658
- **rms_start()**, p. 659

rms_init()

Synopsis

```
#include <rmsLib.h>
void rms_init( void );
```

Library

rmsLib.a

Description

rms_init() initializes the delta queue used by the rate monotonic scheduler (RMS) thread.

rms_init() should be called once before any operations are performed on the delta queue.

Errors

None.

Example

See **rms_start()**, p. 659.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **rms_delq()**, p. 655
- **rms_insq()**, p. 657
- **rms_oldest()**, p. 658
- **rms_start()**, p. 659

Synopsis

```
#include <rmsLib.h>

void rms_insq( d_queue_t *node );
```

Library

rmsLib.a

Description

rms_insq() inserts a node into the delta queue used by the rate monotonic scheduler (RMS) thread.

The delta queue should be initialized using **rms_init()** before items are inserted into the queue. The period field and the key field in the **d_queue_t** structure should be initialized with the thread period of the user-supplied periodic thread. The thread field should contain the thread ID of the periodic thread.

Errors

None.

Example

See **rms_start()**, p. 659.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **rms_delq()**, p. 655
- **rms_init()**, p. 656
- **rms_oldest()**, p. 658
- **rms_start()**, p. 659

rms_oldest()

Synopsis

```
#include <rmsLib.h>
d_queue_t *rms_oldest(void);
```

Library

rmsLib.a

Description

rms_oldest() returns the address of the oldest node in the delta queue used by the rate monotonic scheduler (RMS) thread.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **rms_delq()**, p. 655
- **rms_init()**, p. 656
- **rms_insq()**, p. 657
- **rms_start()**, p. 659

Synopsis

```
#include <rmsLib.h>

pthread_t rms_start( unsigned long tick_rate, int sig );
```

Library

rmsLib.a

Description

rms_start() starts the rate monotonic scheduler (RMS) thread, which manages all user-supplied periodic threads.

tick_rate specifies the period of the scheduler thread, which must equal the number of nanoseconds in the current OS Open tick. *sig* specifies the signal number used to schedule periodic threads.

The RMS scheduling thread has a priority equal to **PTHREAD_PRIO_MAX_NP**.

All user-supplied periodic threads must be running when **rms_start()** is called. Each periodic thread should be started using **rms_wrapper()** as a parameter to the starting **pthread_create()**. **rms_wrapper()** takes as an argument a pointer to a structure containing the following fields:

start_routine	User-supplied periodic thread function
my_arg	Argument passed to the user-supplied periodic thread function
sig_thread	Thread to signal when the user-supplied periodic thread misses a deadline
sig_number	Signal number to send when the user-supplied periodic thread misses a deadline

To stop a periodic thread, send a **SIGKILL** signal to the thread.

rms_wrapper() contains code that support periodic function execution.

If successful, **rms_start()** returns the thread ID of the RMS thread. Otherwise, **rms_start()** returns -1.

Note: Periodic threads managed by the RMS thread must be assigned priorities according to rate monotonic rules.

Errors

None.

rms_start()

Example

The following example starts two periodic threads. The period of the first thread is 250 milliseconds (ms); the period of the second thread is 100 ms.

```
void *thread_one(void *arg);
void *thread_two(void *arg);

struct sched_param param;
pthread_attr attr;
d_queue_t q_one, q_two;
rms_arg_t a_one, a_two;
int j, i;
    •
    •

param.sched_priority=20;
if (pthread_attr_init(&attr)) {
    (void)printf("pthread_attr_init() failed.\n")
    return(-1);
}
(void)pthread_attr_setschedparam(&attr, &param);
(void)rms_init();
q_one.period=q_one.key=250;
a_one.start_routine=thread_one;
a_one.my_arg=(void *)j;
a_one.sig_thread=pthread_self();
a_one.sig_number=SIGUSER2;
if (pthread_create(&q_one.thread, &attr, rms_wrapper, (void *)&a_one)) {
    (void)printf("pthread_create() for thread_one() failed.\n");
    (void)clean_up();
    return(-1);
}
(void)rms_insq(&q_one);
param.sched_priority=21;
(void)pthread_attr_setschedparam(&attr, &param);
q_two.period=q_two.key=100;
a_two.start_routine=thread_two;
a_two.my_arg=(void *)i;
a_two.sig_thread=pthread_self();
a_two.sig_number=SIGUSER2;
if (pthread_create(&q_two.thread, &attr, rms_wrapper, (void *)&a_two)) {
    (void)printf("pthread_create() for thread_two() failed.\n");
    (void)clean_up();
    return(-1);
}
(void)rms_insq(&q_two);
if (sched_thread=rms_start(1000000, SIGUSER1)==(pthread_t)-1) {
```

```
(void)printf("sched_thread() failed.\n");
(void)clean_up();
return(-1);
}
(void)sleep(10000);
:
```

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- pthread_create(), p. 552
- rms_delq(), p. 655
- rms_init(), p. 656
- rms_insq(), p. 657
- rms_oldest(), p. 658

rngBufGet()

Synopsis

```
#include <rngLib.h>

unsigned long rngBufGet( ring_t *ringid, char *buffer,
                        unsigned long count );
```

Library

rngLib.a

Description

rngBufGet() retrieves *count* characters from the ring buffer pointed to by *ringid* and stores the characters in a buffer pointed to by *buffer*.

If successful, **rngBufGet()** returns the actual number of characters stored in the buffer. If *count* equals 0 or the ring buffer is initially empty, **rngBufGet()** returns 0.

Errors

None.

Example

See **sem_open()**, p. 700.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <rngLib.h>
unsigned long rngBufPut( ring_t *ringid, char *buffer,
                        unsigned long count );
```

Library

rngLib.a

Description

rngBufPut() stores *count* characters from the buffer pointed to by *buffer* in the ring buffer pointed to by *ringid*.

If successful, **rngBufPut()** returns the number of characters placed in the ring buffer. If *count* equals 0 or if the ring buffer is initially full, **rngBufPut()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

rngCount()

Synopsis

```
#include <rngLib.h>
unsigned long rngCount( ring_t *ringid );
```

Library

rngLib.a

Description

rngCount() determines the number of characters in the ring buffer pointed to by *ringid*.

If successful, **rngCount()** returns the number of characters in the ring buffer.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <rngLib.h>

ring_t *rngCreate( unsigned long size );
```

Library

rngLib.a

Description

rngCreate() creates a ring buffer; the new ring buffer can store up to *size* characters.

If successful, **rngCreate()** returns a pointer to the new ring buffer. Otherwise, **rngCreate()** returns a NULL pointer.

Errors

[ENOMEM]	Insufficient memory is available to create a new ring buffer.
----------	---

Example

See **sem_open()**, p. 700.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

rngDelete()

Synopsis

```
#include <rngLib.h>
void rngDelete( ring_t *ringid );
```

Library

rngLib.a

Description

rngDelete() destroys the ring buffer pointed to by *ringid*.

The result of using the ring buffer pointer *ringid* after it is destroyed is indeterminate.

rngDelete() returns nothing.

Errors

None.

Example

See **sem_open()**, p. 700.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <rngLib.h>
void rngFlush( ring_t *ringid );
```

Library

rngLib.a

Description

rngFlush() empties the ring buffer pointed to by *ringid*. All characters in the ring buffer are discarded.

rngFlush() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

rnglsEmpty()

Synopsis

```
#include <rngLib.h>
int rnglsEmpty( ring_t *ringid );
```

Library

rngLib.a

Description

rnglsEmpty() returns a non-zero value if the ring buffer pointed to by *ringid* is empty, or 0 if it is not.

Errors

None.

Example

See **sem_open()**, p. 700.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <rngLib.h>
int rngIsFull( ring_t *ringid );
```

Library

rngLib.a

Description

rngIsFull() returns a non-zero value if the ring buffer pointed to by *ringid* is full, or 0 if it is not.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

rootname_add()

Synopsis

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

```
int rootname_add( const char *name, const rootentry_t *entrydata);
```

Library

rtx.o

Description

rootname_add() adds the root directory entry pointed to by *name* to the root directory.

The structure pointed to by *entrydata* can associate user-specific information with *name* in the structure pointed to by *entrydata*. Additionally, **rootname_add()** verifies that *name* consists entirely of characters in POSIX portable name set.

If successful, **rootname_add()** returns 0. Otherwise, **rootname_add()** returns -1 and sets *errno*.

Errors

[ENOENT]	<i>name</i> contains POSIX non-portable characters.
[EEXIST]	A directory entry specified by <i>name</i> already exists.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **driver_install()**, p. 177
- **rootname_delete()**, p. 671
- **rootname_get()**, p. 672
- **rootname_query()**, p. 673

Synopsis

```
#include <sys/ksyscall.h>
int rootname_delete( const char *name );
```

Library

rtx.o

Description

rootname_delete() removes the root directory entry pointed to by *name*.

If successful, **rootname_delete()** returns 0. Otherwise, **rootname_delete()** returns -1 and sets *errno*.

Errors

[ENOENT] The directory entry specified by *name* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **driver_install()**, p. 177
- **rootname_add()**, p. 670
- **rootname_get()**, p. 672
- **rootname_query()**, p. 673

rootname_get()

Synopsis

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

```
int rootname_get( const char *name, rootentry_t *entrydata );
```

Library

rtx.o

Description

rootname_get() returns user-specific information about the root directory entry, specified by *name*, in the structure pointed to by *entrydata*.

If successful, **rootname_get()** returns 0. Otherwise, **rootname_get()** returns -1 and sets *errno*.

Errors

[ENOENT] Directory entry specified by *name* does not exist.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **driver_install()**, p. 177
- **rootname_add()**, p. 670
- **rootname_delete()**, p. 671
- **rootname_query()**, p. 673

Synopsis

```
#include <sys/ksyscall.h>
#include <limits.h>

int rootname_query( const char *name, char *new_name,
                    rootentry_t *entrydata );
```

Library

rtx.o

Description

rootname_query() retrieves consecutive root directory entries.

To retrieve the first entry in the root directory, *name* should point to a null string. *new_name* should point to a string with at least **NAME_MAX** characters. User information on the *new_name* entry is placed in the structure pointed to by *entrydata*.

To retrieve consecutive entries, starting with the first, **rootname_query()** must be called with *name* equal to NULL. Subsequently, after each successful call to **rootname_query()**, the content of *new_name* should be copied to *name* and the call should be issued again.

If successful, **rootname_query()** returns 0. Otherwise, **rootname_query()** returns -1 and sets *errno*.

Errors

[EINVAL]	The directory entry specified by <i>name</i> does not exist.
[ENOENT]	<i>name</i> is the last directory entry.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

References

- **driver_install()**, p. 177
- **rootname_add()**, p. 670
- **rootname_delete()**, p. 671
- **rootname_get()**, p. 672

route()

Synopsis

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

int route( char *cmd );
```

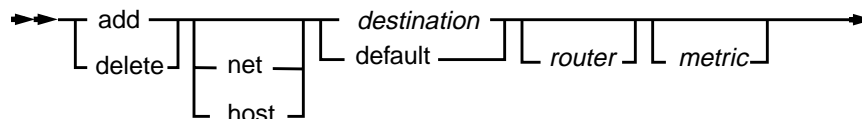
Library

netLib.a

Description

route() allows for manual manipulations of routing tables.

The *cmd* argument points to a string of the following format:



route() distinguishes between routes to hosts and routes to networks by interpreting the Internet address associated with the destination. The Internet address can be specified by a symbolic name or a numeric address. The optional keywords **net** and **host** force the destination to be interpreted as a network or a host, respectively.

If the destination has a local address part of **INADDR_ANY** or if the destination is the symbolic name of a network, then the route is assumed to be to a network; otherwise, it is presumed to be a route to a host. **route()** resolves symbolic names into addresses, using either the **_Tcpip_hosts** array or the network name server. **INADDR_ANY** is defined in **<netinet/in.h>**.

For example, 128.32 is interpreted as **host** 128.0.0.32; 128.32.130 is interpreted as **host** 128.32.0.130; **net** 128.32 is interpreted as 128.32.0.0; and **net** 128.32.130 is interpreted as 128.32.130.0.

Symbolic names specified for a destination or gateway are looked up first as a host name using **gethostbyname()**. If the look up fails, **getnetbyname()** is used to interpret the name as a network name.

destination, which identifies the host or network to which the route is directed, can be specified by symbolic name or numeric address. The keyword **default** can specify all the destinations not defined with another entry.

router, which identifies the gateway to which packets are addressed, can be specified by symbolic name or numeric address.

The keyword **add** adds a route to routing table. If **add** is used, *metric* must be specified. *metric* specifies the number of hops to the destination. The keyword **delete** removes the specified route from the routing table.

If successful, **route()** returns 0. Otherwise, **route()** returns -1.

Errors

None.

Example

See `slip_attach()`, p. 756.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- `gethostbyname()`, p. 265
- `getnetbyname()`, p. 272
- `ifconfig()`, p. 304
- `net_init()`, p. 468
- `ping()`, p. 489

rpc_thread_init()

Synopsis

```
#include <rpc/rpcLib.h>
int rpc_thread_init( void );
```

Library

rpcLib.a

Description

rpc_thread_init() initializes thread private storage for use by RPC library functions.

Before any calls to the RPC library can be made, thread private storage must be initialized.

If successful, **rpc_thread_init()** returns 0. Otherwise, **rpc_thread_init()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

Synopsis

```
#include <rsldLib.h>
```

```
int rsld_setschedparam( int policy, const struct sched_param *sparam )
```

Library

rsldLib.a

Description

rsld_setschedparam() sets the scheduling policy and associated scheduling parameters that are used when the remote source-level debugger creates new threads to debug.

Valid values of **policy** are SCHED_FIFO and SCHED_RR.

The value of `sparam->sched_priority` must be an integer value in the inclusive range of PTHREAD_PRIO_MIN_NP through PTHREAD_PRIO_MAX_NP.

These constants are defined in the file `<pthread.h>`.

If successful, **rsld_setschedparam()** returns 0. Otherwise, **rsld_setschedparam()** returns -1.

Errors

EINVAL	The value of policy is not valid or the value of <code>sparam->sched_priority</code> is not in the proper range.
--------	--

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References:

pthread_setschedparam(), p. 592

rsld_start()

Synopsis

```
#include <rsldLib.h>
int rsld_start( void )
```

Library

rsldLib.a

Description

rsld_start() creates the remote source-level debugger daemon thread.

The daemon thread, which runs in the background, accepts connections from the **rdbx** remote source-level debugger running on a workstation. **rsld_start()** allows the debugger to remotely debug threads running on OS Open over connections made using TCP/IP.

The thread created by **rsld_start()** handles simultaneous connections from multiple copies of **rdbx**, so only one call to **rsld_start()** needs to be made.

If successful, **rsld_start()** returns 0. Otherwise, **rsld_start()** returns -1 and sets *errno*.

Errors

[ENOMEM]	Insufficient memory.
----------	----------------------

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

Synopsis

```
int run( char *parmstr );
```

Library

runLib.a

Description

run() loads and executes an OS Open object module. *parmstr* can either be a NULL terminated string or a NULL pointer. If it is a NULL terminated string, the first word is the name of a dynamically loadable object file with the extension of ".ld" and the rest of the string is passed in as argument strings. If *parmstr* is a NULL pointer, **run()** will read from standard input until it sees the string "exit" or the Ctrl-D character.

The PATH environment string is used to search each directory, separated by colons, for the command name with an extension of ".ld". If no matching files are found, or PATH is not defined, the current working directory is searched.

After the file is loaded, it is executed. The first parameter is the argument count and the next parameter is the array of argument strings.

After the program terminates it is unloaded.

If successful, **run()** returns the value returned from the OS Open object module called by **run()**; otherwise it returns -1.

Example

The following example shows run being called from the OpenShell command prompt to load and run the **cat.ld** program.

```
OS OPEN>run("cat -n README.TXT")
1 a simple ASCII file
OS OPEN>run()
$ cat README.TXT
a simple ASCII file
$ exit
OS OPEN>
```

Errors

None.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

run()

References

- **getopt()**, p. 275
- **ld()**, p. 352
- **setenv_np()**, p. 721
- **shell()**, p. 732

Synopsis

```
#include <stdio.h>

int scanf( const char *format-string, ... );
```

Library

fsLib.a

Description

scanf() reads data from the standard input into locations given by each entry in *argument-list*.

Each argument must be a pointer to a variable of a type that corresponds to a type specifier in *format-string*, which controls interpretation of the input fields.

Note: All options marked with an at sign (@) are not supported in this release of OS Open. Using these options results in an error.

format-string can contain one or more of the following:

- Whitespace characters, as specified by **isspace()** (such as blanks and new lines). A whitespace character causes **scanf()** to read, but not store, all consecutive whitespace characters in the input up to the next character that is not white space. One whitespace character in *format-string* matches any combination of whitespace characters in the input.
- Characters that are not whitespace, except for the percent sign (%). A non-whitespace character causes **scanf()** to read, but not store, a matching non-whitespace character. If the next character in the standard input does not match, **scanf()** ends.
- Format specifications, introduced by a percent sign (%). A format specification causes **scanf()** to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

scanf() reads *format-string* from left to right. Characters outside of format specifications are expected to match the sequence of characters in the standard input; matched characters in the standard input are scanned but not stored. If a character in the standard input conflicts with *format-string*, **scanf()** ends. The conflicting character is left in the standard input as if it had not been read.

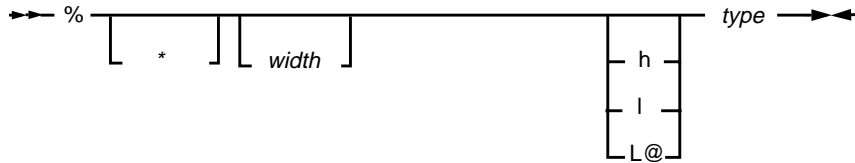
When the first format specification is found, the value of the first input field is converted according to the format specification and stored in the location specified by the entry in *argument-list*. The second format specification converts the second input field and stores it in the second entry in *argument-list*, and so on until the end of *format-string*.

An input field is defined as all characters up to the first whitespace character (space, tab, or new line), up to the first character that cannot be converted according to the format specification, or until *width* is reached, whichever comes

scanf()

first. If there are too many arguments for the format specifications, the extra arguments are ignored. If not enough arguments are given, the results are indeterminate.

A format specification has the following form:



Each field in the format specification is a single character or number signifying a particular format option. *type*, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number. The simplest format specification contains only the percent sign and a *type* character (for example, %s).

Each field in the format specification is subsequently described in detail. If a percent sign (%) is followed by a character that has no meaning as a format control character, **scanf()** returns an error. The only exception is the percent sign(%). For example, to specify a percent sign, use "%%".

An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified *type*. The field is scanned but not stored.

width, a positive decimal integer, controls the maximum number of characters to be read from the standard input. No more than *width* characters are converted and stored at the corresponding argument. Fewer than *width* characters are read if a whitespace character (space, tab, or new line), or a character that cannot be converted according to the given format, occurs before *width* is reached.

The optional prefix **l** shows use of the long version of the following *type*; the optional prefix **h** indicates that the short version is used. The corresponding argument should point to a long object (for the **l** prefix), a long double object (for the **L** prefix), or a short object (for the **h** prefix). The **l** and **h** modifiers can be used with the **d**, **o**, **x**, **X**, **i**, and **u** characters. The **L** modifier can be used with the **e**, **f**, **g**, **E**, and **G** characters. The **l** and **h** modifiers are ignored if they are specified for any other character *type*. The *type* characters and their meanings are in the following table:

Table 44. Types Used by scanf()

Character	Type of Input Expected	Type of Argument
d	Decimal integer	Pointer to int
o	Octal integer	Pointer to unsigned int

Table 44. Types Used by scanf()

Character	Type of Input Expected	Type of Argument
x X	Hexadecimal integer	Pointer to unsigned int
i	Decimal, hexadecimal, or octal integer	Pointer to int
u	Unsigned decimal integer	Pointer to unsigned int
e f g E G	Floating-point value consisting of an optional sign (+ or –); a series of one or more decimal digits possibly containing a decimal point; and an optional exponent (“e” or “E”) followed by a possibly signed integer value	Pointer to float
c	Character; whitespace characters that are ordinarily skipped are read when c is specified	Pointer to char large enough for input field
s	String	Pointer to character array large enough for input field plus a terminating NULL character (\0), which is automatically appended
n	No input read from <i>stream</i> or buffer	Pointer to int , into which is stored the number of characters successfully read from the <i>stream</i> or buffer up to that point in the call to scanf()
p	Pointer to void converted to series of characters	Pointer to void
lc@	Multibyte character constant	Pointer to wchar_t
ls@	Multibyte string constant	Pointer to wchar_t string
[] [^]	String	Pointer to character array large enough to input field plus a terminating NULL character (\0), which is automatically appended

To scan for certain characters, use bracket ([]) characters. The scan set comprises the characters between the two brackets. If the first character in the scan set is a caret (^), characters outside the scan set are searched for. A closing bracket (]) is included in the scan set by making it the first character in the scan set following the caret (^), if used.

scanf()

A range is entered as a scan set by entering the lowest character and highest character separated by a dash(-), such as “[a-z].” If the scan set is not in the format for a range, only the characters within the brackets comprise the scan set.

To store a string without storing an ending NULL(\0), use the specification `%nc`, where *n* is a decimal integer. In such a case, the **c** type character means that the argument is a pointer to a character array. The next *n* characters are read from the input stream into the specified location, and no NULL (\0) is added.

The input for a `%x` format specifier is interpreted as a hexadecimal number.

scanf() scans each input field, character by character. **scanf()** may stop reading a particular input field before the function reaches a space character, when *width* is reached, or when the next character cannot be converted as specified. When a conflict occurs between the specification and the input character, the next input field begins at the first unread character. The conflicting character, if any, is considered unread and is the first character of the next input field or the first character in subsequent read operations on the standard input.

scanf() returns the number of fields successfully converted and assigned. The return value does not include fields that were read but not assigned. **scanf()** returns **EOF** for an attempt to read at end-of-file. If no fields were assigned, **scanf()** returns 0.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

The following example takes a date entered and converts it to the number of seconds since January 1, 1970 00:00:00.

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

void date(void)
{
    char c = 'y';
    char buff[50];
    struct tm t;
    time_t seconds;
    int days_month[12] =
        {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    while(c == 'y')
    {
        /* Reset February to 28 */
        days_month[1] = 28;
        memset(&t, 0, sizeof(struct tm));
        /* Get date from stdin */
        printf("Enter a date with the format: MM/DD/YY\n");
        scanf("%d/%d/%d", &t.tm_mon, &t.tm_mday, &t.tm_year);
        /* Check for valid month */
        if((t.tm_mon < 1) || (t.tm_mon > 12))
        {
            printf("Month must be a number between 1 and 12\n");
            continue;
        }
        /* subtract one from month because time functions use */
        /* month numbers 0–11, 0 is January */
        t.tm_mon--;
        /* Check for a valid year */
        if((t.tm_year < 70) || (t.tm_year > 99))
        {
            printf("The year must be between 1970 and 1999\n");
            continue;
        }
        /* If leap year set February to 29 days */
        if(t.tm_year % 4 == 0)
            days_month[1] = 29;
        /* Check for valid day of the month */
        if((t.tm_mday < 1) ||
            (t.tm_mday > days_month[t.tm_mon]))
        {
            printf("Day of the month must be between 1 and %d\n",
```

scanf()

```
        days_month[t.tm_mon]);
    continue;
}
/* Get the number of seconds */
seconds = mktime(&t);
/* Format date entered to Month Day, Year */
strftime(buf, 50, "%B %d, %Y", &t);
/* Print out answer */
printf("Number of seconds since January 1, 1970 00:00:00\n");
printf("to %s: %d\n", buf, seconds);
/* Check to see if user wants to quit */
printf("Enter 'y' to continue, 'n' to quit.\n");
fflush(stdin);
valid_entry = 0;
while (!valid_entry)
{
    c = getchar();
    if (c == 'y' || c == 'n')
        valid_entry = 1;
}
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.6.4
- **fs_init()**, p. 231
- **fscanf()**, p. 234
- **printf()**, p. 500
- **sscanf()**, p. 768

Synopsis

```
#include <sched.h>

void sched_getmask( unsigned long *mask );
```

Library

rtxLib.a

Description

sched_getmask() retrieves the scheduling mask and stores it in a variable pointed to by *mask*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **pthread_attr_setschedparam()**, p. 524
- **pthread_create()**, p. 552
- **sched_setmask()**, p. 688

sched_setmask()

Synopsis

```
#include <sched.h>

int sched_setmask( unsigned long mask );
```

Library

rtxLib.a

Description

sched_setmask() sets scheduling mask to *mask*.

mask specifies the scheduling mask to be used in dispatching threads. Priority that is masked by **sched_setmask()** will not be eligible for dispatch unless there is no other ready thread in the system. The least significant bit in *mask* selects priority `PTHREAD_PRIO_MIN_NP` and the most significant bit selects priority `PTHREAD_PRIO_MAX_NP`. A zero bit specifies that the selected priority should be masked off.

This facility can be used to dynamically change execution characteristics for groups of threads without individually altering thread parameters.

sched_setmask() always returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **pthread_attr_setschedparam()**, p. 524
- **pthread_create()**, p. 552
- **sched_getmask()**, p. 687

Synopsis

```
#include <sched.h>
void sched_yield( void );
```

Library

rtxLib.a

Description

sched_yield() causes the calling thread to yield its processor to another ready thread of the same priority.

The calling thread's execution is halted and the thread is made runnable so it may be rescheduled.

sched_yield() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1b, 1993

schednetisr()

Synopsis

```
#include <net/netisr.h>
schednetisr( int isr_type );
```

Library

None. This is a macro defined in **<net/netisr.h>**.

Description

schednetisr() schedules a software network interrupt that signals the arrival of new Internet Protocol (IP) packets in the common input queue.

OS Open supports only one software network interrupt, so *isr_type* should be set to **NETISR_IP**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

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

int select( int numfiles, int *readmask, int *writemask,
            int *excpmask, struct timespec *timeout );
```

Library

devLib.a

Description

select() allows an application to query a file (usually a character or block special file) to determine whether there is data to read, space to write, or any outstanding exception conditions.

select() may block, specifying a timeout, until one of the preceding conditions is true. *numfiles* specifies the number of significant bits in the following masks. For example, to determine status on file descriptors 3, 7, 2, and 9, *numfiles* is set to 10. Ten bits, corresponding to file descriptors 0 through 9, are examined in the following masks.

readmask, *writemask*, and *excpmask* each point to an **int** treated as an array of bits. If the bit index of the corresponding mask is set, that file descriptor is checked for the corresponding condition. The bit corresponding to file descriptor *n* in a mask is set by turning on the bit $(1 \ll n)$ in the mask.

readmask indicates that a file has data to read. *writemask* indicates that a device driver has space to transfer data using **write()**. *excpmask* tests for an outstanding exception condition. When **select()** completes, the bits in the three masks are modified to reflect the conditions that were true at the time of the call.

timeout controls the operation of **select()** if no tested condition is true. If *timeout* points to a **timespec** structure containing 0s, **select()** polls for the desired status and returns immediately. If *timeout* is NULL, **select()** blocks indefinitely and waits for at least one condition to be true. Otherwise, **select()** waits for the time specified by *timeout* or until at least one condition is true.

If successful, **select()** returns the number of file descriptors meeting the test criteria specified in the masks and sets the bits in the masks to the conditions that were true at the time of the call. If the time specified by *timeout* expires, **select()** returns 0. Otherwise, **select()** returns -1 and sets *errno*.

Errors

[EBADF]	A file descriptor is not valid.
[EINVAL]	A parameter is not valid.

select()

Example

The following example monitors the standard input, standard output, and standard error for exception conditions. If an exception condition is found on one of those three files, it is closed and reopened for the async device driver.

```
#include <sys/select.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <stddef.h>
#include <sys/devLib.h>
#include <sys/devDriver.h>
#include <sys/ioctl.h>
#include <asyncLib.h>

#define Standardin 0x1
#define Standardout 0x2
#define Standarderr 0x4

int oldstate
void iomonitor(void)
{
    int readmask = 0, writemask = 0, excpmask = 0x7;
    int rc = 0;
    while(rc >= 0)
    {
        rc = select(3, &readmask, &writemask, &excpmask, NULL);
        if(excpmask & Standardin)
        {
            pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
            close(0);
            open("dev/tty0", O_RDONLY, asyncParityNone, asyncParityOdd,
                asyncStopBits1, asyncDataBits8, 9600);
            pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
        }
        if(excpmask & Standardout)
        {
            pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
            close(1);
            open("dev/tty0", O_WRONLY, asyncParityNone, asyncParityOdd,
                asyncStopBits1, asyncDataBits8, 9600);
            pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
        }
        if(excpmask & Standarderr)
        {
            pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
```

```
        close(2);
        open("dev/tty0", O_WRONLY, asyncParityNone, asyncParityOdd,
            asyncStopBits1, asyncDataBits8, 9600);
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
    }
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **read()**, p. 633
- **write()**, p. 960

select_notify()

Synopsis

```
#include <sys/devDriver.h>
int select_notify( file_block_t *fb, int flags );
```

Library

devLib.a

Description

select_notify() allows a device driver to signal a previously requested notification of a **select()** status change.

When the device-specific **select()** is invoked, a flag indicates whether asynchronous notification is requested if the desired condition becomes true. The device driver signals that change using **select_notify()**.

fb should point to the file block passed using the original **select()** for the request. *flags* is set to the current read, write, and exception status using the **SEL_READ**, **SEL_WRITE**, and **SEL_EXCP** constants, which are defined in the file **<sys/select.h>**.

The device driver needs to call **select_notify()** only once for any recorded status change. If multiple **select()** functions are outstanding, the device subsystem reissues device-specific **select()** functions if necessary.

If successful, **select_notify()** returns 0. Otherwise, **select_notify()** returns -1 and sets *errno*.

Errors

[EINVAL]	Internal error.
----------	-----------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **select()**, p. 691

Synopsis

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

int select_redrive( file_block_t *fblk, int fd, int events, int *ret_events );
```

Library

devLib.a

Description

select_redrive() is used internally by device drivers to redrive a device-specific **select()** to another device.

For example, an implementation of a tty driver may need to issue **select()** to an underlying terminal hardware driver. **select_redrive()** is to be called from the device-specific **select()** of the original driver.

fblk is set to the file block pointer value supplied to the original **select()**. *fd* is set to the file descriptor of the target device. *events* should be set to the desired read, write, and exception status using the **SEL_ASYNC**, **SEL_READ**, **SEL_WRITE**, and **SEL_EXCP** constants, which are defined in the file **<sys/select.h>**.

The integer pointer to *ret_events* is set to the current state of *fd* by **select_redrive()**. Unlike **select()**, this function does not block if asynchronous notification is specified. Internal flags are set to retest the status at the original **select()** level when the appropriate **select_notify()** is called by the underlying device drivers.

If successful, **select_redrive()** returns 0. Otherwise, **select_redrive()** returns -1 and sets *errno*.

Errors

[EINVAL]	Internal error.
----------	-----------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **select()**, p. 691
- **select_notify()**, p. 694

sem_close()

Synopsis

```
#include <semaphor.h>
int sem_close( sem_t *sem );
```

Library

rtxLib.a

Description

sem_close() indicates that OS Open is finished using the semaphore named by *sem*.

After **sem_close()** is called for a semaphore, the semaphore is closed for all threads in OS Open. If **sem_close()** is called for an unnamed semaphore (one created by **sem_init()**), the results are undefined. If the semaphore was not removed using a successful **sem_unlink()**, **sem_close()** does not affect the state of the semaphore.

If the semaphore was marked as destructible by a previous **sem_unlink()**, the semaphore is not destroyed until all open descriptors associated with the semaphore have been closed. Using a semaphore descriptor after returning from **sem_close()** and before calling **sem_open()** produces erroneous results.

If successful, **sem_close()** returns 0. Otherwise, **sem_close()** returns -1 and sets *errno*.

Errors

[EINVAL] *sem* is not a valid semaphore descriptor.

Example

See **sem_open()**, p. 700.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §11.2.4
- **sem_init()**, p. 699
- **sem_open()**, p. 700
- **sem_unlink()**, p. 708

Synopsis

```
#include <semaphor.h>
int sem_destroy( sem_t *sem );
```

Library

rtxLib.a

Description

sem_destroy() destroys the unnamed semaphore descriptor identified by *sem*.

Only a semaphore that was created using **sem_init()** can be destroyed using **sem_destroy()**. The effect of calling **sem_destroy()** with a named semaphore is undefined.

Using the semaphore descriptor after returning from **sem_destroy()** and before calling **sem_init()** produces erroneous results.

If successful, **sem_destroy()** returns a value of 0. Otherwise, **sem_destroy()** returns -1 and sets *errno*.

Errors

[EINVAL] *sem* is not a valid semaphore descriptor.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §11.2.2
- **sem_init()**, p. 699
- **sem_open()**, p. 700

sem_getvalue()

Synopsis

```
#include <semaphor.h>

int sem_getvalue( sem_t *sem, int *sval );
```

Library

rtxLib.a

Description

sem_getvalue() retrieves the value of the semaphore pointed to by *sem* and updates the location pointed to by *sval*.

sem_getvalue() does not change the state of the semaphore.

If the semaphore is locked, the value returned by **sem_getvalue()** is either 0 or a negative number whose absolute value represents the number of threads waiting for the semaphore.

If successful, **sem_getvalue()** returns 0. Otherwise, **sem_getvalue()** returns -1 and sets *errno*.

Errors

[EINVAL] *sem* is not a valid semaphore descriptor.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §11.2.8
- **sem_post()**, p. 705
- **sem_trywait()**, p. 707
- **sem_wait()**, p. 709

Synopsis

```
#include <semaphor.h>

int sem_init( sem_t *sem, int pshared, unsigned int value );
```

Library

rtxLib.a

Description

sem_init() initializes an unnamed semaphore descriptor named by *sem*.

The semaphore is created with an initial *value*, which must be 0 or positive. (A negative value implies that at least one thread is blocked on the semaphore.) Valid semaphore values cannot be greater than **SEM_VALUE_MAX**, which is defined in the file **<limits.h>**.

After the semaphore descriptor has been initialized, it can be used on subsequent calls to **sem_wait()**, **sem_trywait()**, and **sem_post()**. The semaphore remains usable until it is destroyed.

The *pshared* attribute is ignored.

If successful, **sem_init()** returns 0. Otherwise, **sem_init()** returns -1 and sets *errno*. The semaphore descriptor identified by *sem* is unchanged.

Errors

[EINVAL]	<i>value</i> is not valid.
[ENOSPC]	Insufficient memory is available to initialize the semaphore descriptor.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §11.2.1
- **sem_destroy()**, p. 697
- **sem_post()**, p. 705
- **sem_trywait()**, p. 707
- **sem_wait()**, p. 709

sem_open()

Synopsis

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

sem_t *sem_open( const char *name, int oflag, ... );
```

Library

rtxLib.a

Description

sem_open() establishes a connection between a named semaphore and OS Open.

After a call to **sem_open()** using the semaphore name pointed to by *name*, the process may refer to the semaphore associated with *name* using the address returned by this call. The semaphore may be used in subsequent calls to **sem_wait()**, **sem_timedwait()**, **sem_trywait()**, and **sem_post()**. The semaphore remains usable by the process until the semaphore is closed by a successful **sem_close()**.

oflag controls whether the semaphore is created or accessed by **sem_open()**. The flags are defined in **<fcntl.h>**.

The following flag bits may be set in *oflag*.

O_CREAT This flag creates a semaphore if it does not already exist. If **O_CREAT** is set and the semaphore exists, **O_CREAT** has no effect, except as noted under **O_EXCL**, below. Otherwise, **sem_open()** creates a named semaphore.

When the **O_CREAT** flag is set, the third argument *mode*, of type **mode_t**, is required. Although *mode* is ignored by this function, it is required for POSIX portability.

sem_open() also uses the fourth argument, *value*, an **unsigned int**. The semaphore is created with the initial value *value*. Valid initial values cannot be greater than **SEM_VALUE_MAX**.

After the semaphore named *name* is created by **sem_open()** with the **O_CREAT** flag, other threads may connect to the semaphore by calling **sem_open()** using the same value of *name*.

O_EXCL If **O_EXCL** and **O_CREAT** flags are set, the **sem_open()** function will fail if the semaphore *name* exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist is atomic with respect to other threads executing the **sem_open()** function call with **O_EXCL** and **O_CREAT** set.

If flags other than **O_CREAT** or **O_EXCL** are specified, they are ignored.

name points to a string containing the path name of a semaphore. The path name pointed to by *name* must contain fewer than **NAME_MAX** characters from the portable file name character set. The constant **NAME_MAX** is defined in the file **<limits.h>**. The name may include slash characters, although they have no special significance. The current directory does not affect the interpretation of the name. If multiple threads successfully call **sem_open()** with the same *name* attribute, the same semaphore address is returned for each successful call, provided that there have been no calls to **sem_unlink()** for this semaphore.

If successful, **sem_open()** returns the address of the semaphore. Otherwise, **sem_open()** returns *(sem_t *) - 1* and sets *errno*.

Errors

[EEXIST]	O_CREAT and O_EXCL flags were set and the named semaphore already exists.
[EINVAL]	O_CREAT was specified in <i>oflag</i> , and <i>value</i> is not valid, or the object pointed to by <i>name</i> contains characters not in the portable file name character set.
[ENAMETOOLONG]	The length of the string <i>name</i> exceeds NAME_MAX .
[ENOENT]	The semaphore that <i>name</i> refers to does not exist and the O_CREAT flag is not set.
[ENOSPC]	Insufficient memory is available to create a semaphore.

Example

The following example contains some functions for a read/write device using ring buffers. The ring buffers are protected by semaphores.

```
#include <semaphor.h>
#include <sys/devLib.h>
#include <sys/devDriver.h>
#include <sys/ksyscall.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include <rngLib.h>
#include <fcntl.h>

typedef struct rwdds
{
    ring_t *recv_buf; /* ring buffer for receiving data */
    sem_t *recv_buf_sem; /* semaphore protecting the receive buffer */
    ring_t *send_buf; /* send buffer for writing data */
}
```

sem_open()

```
sem_t *send_buf_sem; /* semaphore protecting the write buffer */
} DDS;
int device_init(int recv_buf_size, int send_buf_size, void **dds)
{
    char *rs = {"recv_sem"};
    char *ss = {"send_sem"};
    DDS *device;

    /* allocate device structure block */
    device = (DDS *)malloc(sizeof(DDS));
    if(device == NULL)
        return(-1);
    /* Initialize the fields in the device structure block */
    device->recv_buf = rngCreate(recv_buf_size);
    device->recv_buf_sem = sem_open(rs, O_CREAT, 0, 1);
    device->send_buf = rngCreate(send_buf_size);
    device->send_buf_sem = sem_open(ss, O_CREAT, 0, 1);
    if((device->recv_buf_sem < 0) || (device->send_buf_sem < 0) ||
        (device->recv_buf == 0) || (device->send_buf == 0))
    {
        if((int)device->recv_buf_sem != -1) {
            sem_unlink(rs);
            sem_close(device->recv_buf_sem);
        }
        if((int)device->send_buf_sem != -1) {
            sem_unlink(ss);
            sem_close(device->send_buf_sem);
        }
        if((int)device->recv_buf != 0)
            rngDelete(device->recv_buf);
        if((int)device->send_buf != 0)
            rngDelete(device->send_buf);
        return(-1);
    }
    *dds = (void *)device;
    /* Associate the name "read/write_device" with the device pointer */
    package_install("read/write_device", (void *)device, NULL);
    return(0);
}
int device_read(file_block_t *fb, void *buffer, size_t length)
{
    DDS *device = (DDS *)fb->driver_specific;
    size_t count = 0;
    sigset_t set;
    char *wp = (char *)buffer;
    size_t rc;
```

```
int sig;
/* lock receive buffer semaphore */
rc = sem_wait(device->recv_buf_sem);
if(rc != 0)
{
    /* lock failed because of pending signal */
    if(*pthread_errno_np() == EINTR) {
        /* consume the signal and try locking it again */
        sigemptyset(&set);
        sigpending(&set);
        pthread_sigmask(SIG_SETMASK, &set, NULL);
        sigwait(&set, &sig);
        rc = sem_wait(device->recv_buf_sem);
        if(rc != 0)
            return(-1);
    }
    else
        return(-1);
}
/* receive in the data */
while(count < length)
{
    rc = rngBufGet(device->recv_buf, wp, length - count);
    wp += rc;
    count += rc;
    if(rngIsEmpty(device->recv_buf) > 0)
        break;
}
/* Unlock the semaphore and return */
sem_post(device->recv_buf_sem);
return(count);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

sem_open()

References

- IEEE Draft Standard 1003.4/D13, §11.2.3
- **select_notify()**, p. 694
- **sem_close()**, p. 696
- **sem_post()**, p. 705
- **sem_trywait()**, p. 707
- **sem_unlink()**, p. 708
- **sem_wait()**, p. 709

Synopsis

```
#include <semaphor.h>
int sem_post( sem_t *sem );
```

Library

rtxLib.a

Description

sem_post() increments the counting semaphore pointed to by *sem*.

If the incremented value of the semaphore is negative or 0, the oldest highest-priority thread blocked on the semaphore is unblocked.

If the semaphore pointed to by *sem* has a count of **SEM_VALUE_MAX**, **sem_post()** has no effect and returns without error. **SEM_VALUE_MAX** is defined in the file **<limits.h>**.

If successful, **sem_post()** returns 0. Otherwise, **sem_post()** returns -1 and sets *errno*.

Errors

[EINVAL] *sem* does not refer to a valid semaphore.

Example

See **sem_open()**, p. 700.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §11.2.7
- **sem_trywait()**, p. 707
- **sem_wait()**, p. 709

sem_timedwait()

Synopsis

```
#include <semaphor.h>

int sem_timedwait( sem_t *sem, const struct timespec *timeout );
```

Library

rtxLib.a

Description

sem_timedwait() decrements the counting semaphore pointed to by *sem*. **sem_timedwait()** times out if the time interval specified in *timeout* has elapsed while the thread was blocked on the semaphore. If the semaphore times out, the semaphore is incremented.

If the decremented value of the semaphore becomes negative, the calling thread is blocked for, at most, the time interval specified in *timeout*.

If successful, **sem_timedwait()** returns 0. Otherwise, **sem_timedwait()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>sem</i> does not point to a valid semaphore.
[EINTR]	An unblocked signal is pending on OS Open or the thread.
[ETIMEDOUT]	Time specified in <i>timeout</i> has elapsed while the thread was blocked on the semaphore.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **sem_post()**, p. 705
- **sem_wait()**, p. 709

Synopsis

```
#include <semaphor.h>
int sem_trywait( sem_t *sem );
```

Library

rtxLib.a

Description

sem_trywait() decrements the counting semaphore identified by *sem* if the decremented value of the semaphore would be non-negative.

Otherwise, the call returns without affecting the value of the semaphore.

If successful, **sem_trywait()** returns 0. Otherwise, **sem_trywait()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>sem</i> does not refer to a valid semaphore.
[EAGAIN]	The semaphore was already locked, (that is, the semaphore count was less than or equal to 0) so it cannot be immediately locked by sem_trywait() .

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §11.2.6
- **sem_post()**, p. 705

sem_unlink()

Synopsis

```
#include <semaphor.h>
int sem_unlink( const char *name );
```

Library

rtxLib.a

Description

sem_unlink() marks a named semaphore as destructible.

The semaphore is specified by the NULL terminated string *name*.

If there are no semaphore descriptors associated with the specified semaphore, the semaphore is destroyed. Otherwise, destruction of the semaphore is postponed until all semaphore descriptors associated with the semaphore have been closed by calls to **sem_close()**.

Once a semaphore is marked as destructible, subsequent calls to **sem_open()** specifying the same *name* refer to a new semaphore of the same name.

If successful, **sem_unlink()** returns 0. Otherwise, **sem_unlink()** returns -1 and sets *errno*.

Errors

[ENOENT] The semaphore specified by *name* does not exist.

Example

See **sem_open()**, p. 700.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §11.2.5
- **select_notify()**, p. 694
- **sem_close()**, p. 696
- **sem_open()**, p. 700

Synopsis

```
#include <semaphor.h>
int sem_wait( sem_t *sem );
```

Library

rtxLib.a

Description

sem_wait() decrements the counting semaphore pointed to by *sem*.
If the decremented value of the semaphore becomes negative, the calling thread is blocked.
If successful, **sem_wait()** returns 0. Otherwise, **sem_wait()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>sem</i> does not point to a valid semaphore.
[EINTR]	An unblocked signal is pending on OS Open or the thread.

Example

See **sem_open()**, p. 700.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4/D13, §11.2.6
- **sem_post()**, p. 705

semaphore_dump()

Synopsis

```
#include <kadtLib.h>
int semaphore_dump( sem_t semid, char *buf );
```

Library

kadtLib.a

Description

semaphore_dump() displays information about the semaphore identified by *semid*.

If *buf* is NULL, the output is generated by **bprintf()**. Otherwise, the output is stored in the buffer pointed to by *buf*.

semaphore_dump() returns -1 if *semid* is not a valid semaphore ID. Otherwise, **semaphore_dump()** returns 0.

Errors

None.

Example

The following shows a sample dump of semaphore 0x426e78.

```
OS OPEN>semaphore_dump(0x426e78)
```

```
-----
Semaphore ID: 0x426e78
```

```
Name:
```

```
Count: 0 Num.Opens: 1 State: INIT
```

```
Waiting threads:
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84

Synopsis

```
#include <kadtLib.h>
int semaphore_list( char **bufp );
```

Library

kadtLib.a

Description

semaphore_list() provides formatted information about all semaphores.

The information is part of the **kda_dump()** display.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

semaphore_list() returns 0.

Errors

None.

Example

The following shows a sample listing of all semaphores.

```
OS OPEN>semaphore_list()
Semaphore  Name                                Count
0x426e78                                         0
0x426ebc                                         1
0x469158                                         0
0x46919c                                         0
0x46db0c  Network_name_sem                     1
0x46db84  Network_netw_sem                     1
0x46dbfc  Network_prot_sem                     1
0x46dc74  Network_serv_sem                       1
0x4aeb98  Tcip_sem_01                           -1
0x4bcd0  Async_sem_14                             0
0x4bce60  Async_sem_13                             0
0x4bcd0  Async_sem_12                             1
0x4bcf40  Async_sem_11                             1
0x4bd900  Async_sem_04                             0
0x4bd970  Async_sem_03                             0
0x4bd9e0  Async_sem_02                             1
0x4bda50  Async_sem_01                             1
0x4bef3c  /sem/devselect.sem                       -1
----- Total Number of Semaphores: 18 -----
```

semaphore_list()

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84
- **kda_dump()**, p. 349

Synopsis

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

int send( int s, const void *msg, int len, int flags );
```

Library

netLib.a

Description

send() allows an application program to send messages using a connected socket named by *s*.

send() is equivalent to **sendto()**, except that no destination is specified. *len* specifies the length of the message. *msg* points to the buffer that holds the message. If the message is too long to pass through the underlying protocol, the error **[EMSGSIZE]** is returned and the message is not transmitted.

If the sending socket has no space to hold the message to be transmitted, **send()** blocks the message unless the socket is in a non-blocking I/O mode.

flags controls message transmission. *flags* is formed by applying a logical OR to one or both of the following constants, which are defined in the file **<sys/socket.h>**:

MSG_DONTROUTE	Send without using routing tables.
MSG_OOB	Process out-of-band data. The underlying protocol must support out-of-band data, for example, TCP.

If successful, **send()** returns the number of characters sent. Otherwise, **send()** returns **-1** and sets *errno*.

Errors

[EBADF]	The socket descriptor <i>s</i> is not valid.
[ENOTSOCK]	The socket descriptor <i>s</i> references a file, not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking, and the send operation would block.
[EMSGSIZE]	The message is too large to be sent all at once, as the socket requires.
[ENOBUFS]	OS Open could not allocate an internal buffer for the message. The output queue for a network interface is full.
[EINTR]	A signal interrupted send() before all data was transmitted.

send()

Example

The following routine sends a character on a connected socket when called. When the routine is first called, *connect_flag* should be set to a non-zero value to cause a socket connection to be made. On subsequent calls, setting *connect_flag* to 0 causes the character to be sent on the socket established on the first call.

```
#define BYTE_ORDER BIG_ENDIAN
#include <types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <machine/endian.h>
#include <arpa/nameser.h>

int s;
struct sockaddr_in sa;

int ex_send(char * sendchr, int urgent, int connect_flag)
{
    int rc = 0;

    if (connect_flag) {
        sa.sin_addr.s_addr=0x0943A4F9; /* 9.67.164.249 */
        sa.sin_family=AF_INET;
        sa.sin_port=htons(50003);
        s=socket(AF_INET, SOCK_STREAM, 0);
        if (s<0) {
            (void)printf("socket() call failed\n");
            return(-1);
        }
        if (connect(s, (struct sockaddr *)&sa, sizeof(struct sockaddr)) < 0) {
            (void)printf("connect() call failed\n");
            return(-1);
        }
    }
    printf("entering send, socket number %d, sending character %c\n", s,
        *sendchr);
    if (urgent) {
        rc=send(s, sendchr, 1, MSG_OOB);
    } else {
        rc=send(s, sendchr, 1, 0);
    }
    return(rc);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.11
- **getsockname()**, p. 295
- **read()**, p. 633
- **recv()**, p. 640
- **select()**, p. 691
- **sendmsg()**, p. 716
- **socket()**, p. 760
- **write()**, p. 960

sendmsg()

Synopsis

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

int sendmsg( int s, const struct msghdr *msg, int flags );
```

Library

tcpipLib.a

Description

sendmsg() sends messages on a connected or unconnected socket, named by *s*, using the **msghdr** message structure.

To broadcast on a socket, the application program must first call **setsockopt()** using the **SO_BROADCAST** option to gain broadcast permissions.

sendmsg() supports no more than 15 message elements.

msg points to the **msghdr** message structure containing the message to be sent; the structure decreases the number of directly supplied parameters. The **msghdr** structure is defined in the file **<sys/socket.h>**.

```
struct msghdr {
    caddr_t      msg_name;
    u_int        msg_namelen;
    struct iovec *msg_iov;
    u_int        msg_iovlen;
    caddr_t      msg_control;
    u_int        msg_controllen;
    int          msg_flags;
}
```

msg_name and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be a null pointer if no names are desired or required. *msg_iov* and *msg_iovlen* describe the scatter gather locations. *msg_control* of length *msg_controllen* is a buffer for other protocol-related messages or other miscellaneous data. *msg_flags* is set on return in a way that may include some of the values specified for the *flags* argument to **recv()**.

flags controls message transmission. *flags* is formed by applying a logical OR to one or both of the following constants, which are defined in the file **<sys/socket.h>**:

MSG_DONTROUTE	Send without using routing tables.
MSG_OOB	Process out-of-band data. The underlying protocol must support out-of-band data, for example, TCP.

If successful, **sendmsg()** returns the number of characters sent. Otherwise, **sendmsg()** returns -1 and sets *errno*.

Errors

[EBADF]	The socket descriptor <i>s</i> is not valid.
[ENOTSOCK]	The socket descriptor <i>s</i> references a file, not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking, and the receive operation would block.
[EMSGSIZE]	The message is too large to be sent all at once, as the socket requires.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.11
- **getsockopt()**, p. 296
- **read()**, p. 633
- **recv()**, p. 640
- **select()**, p. 691
- **sendto()**, p. 718
- **socket()**, p. 760
- **recvmsg()**, p. 645
- **write()**, p. 960

sendto()

Synopsis

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

int sendto( int s, const void *msg, int msglen, int flags,
            const struct sockaddr *to, int tolen );
```

Library

tcpipLib.a

Description

sendto() allows an application program to send messages through an unconnected socket named by *s* by specifying a destination address.

To broadcast on a socket, first call **setsockopt()** using the **SO_BROADCAST** option to gain broadcast permissions.

to points to the address of the target socket. *msglen* specifies the message length. *tolen* specifies the size of *to*.

If the message is too long to pass through the underlying protocol, the error **[EMSGSIZE]** is returned, and the message is not transmitted. If the sending socket cannot store the message, **sendto()** blocks the message unless the socket is in non-blocking I/O mode. *msg* points to the address containing the message.

flags controls message transmission. *flags* is formed by applying a logical OR to one or both of the following constants, which are defined in the file **<sys/socket.h>**:

MSG_DONTROUTE	Send without using routing tables.
MSG_OOB	Process out-of-band data. The underlying protocol must support out-of-band data, for example, TCP.

If successful, **sendto()** returns the number of characters sent. Otherwise, **sendto()** returns **-1**, and sets *errno*.

Errors

[EBADF]	The socket descriptor <i>s</i> is not valid.
[ENOTSOCK]	The socket descriptor <i>s</i> references a file, not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking, but the receive operation would block.
[EMSGSIZE]	The message is too large to be sent all at once, as the socket requires.
[ENOBUFS]	OS Open could not allocate an internal message buffer. The output queue for a network interface was full.
[EOPNOTSUPP]	The operation is not supported.

[EHOSTUNREACH] No route to host.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.11
- **getsockname()**, p. 295
- **read()**, p. 633
- **recv()**, p. 640
- **select()**, p. 691
- **sendmsg()**, p. 716
- **socket()**, p. 760
- **write()**, p. 960

setbuf()

Synopsis

```
#include <stdio.h>

void setbuf( FILE *stream,
             char *buffer );
```

Library

fsLib.a

Description

setbuf() controls buffering for the stream pointed to by *stream*.

stream must refer to an open file that has not yet been read or written. If *buffer* is NULL, the stream is not buffered. If not, *buffer* must point to a character array of length **BUFSIZ**, which is defined in the file **<stdio.h>**. OS Open uses *buffer* instead of the default system-allocated buffer for the stream.

setbuf() returns nothing.

setvbuf() is more flexible than **setbuf()**.

Note: The file system must have been initialized using **fs_init()**.

Errors

None.

Example

See **fs_init()**, p. 231.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.5.5
- **fclose()**, p. 190
- **fflush()**, p. 197
- **fopen()**, p. 210
- **fs_init()**, p. 231
- **setvbuf()**, p. 730

Synopsis

```
#include <stdlib.h>
int setenv_np( const char *var );
```

Library

cLib.a

Description

setenv_np() defines a new environment variable or redefines an existing environment variable.

If the string pointed to by *var* does not have the following syntax, **setenv_np()** returns an error:

variable = definition

If successful, **setenv_np()** returns 0. Otherwise, **setenv_np()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>var</i> points to a string that is not valid.
[ENOMEM]	Insufficient memory is available to define or redefine the environment variable.

Example

See **fs_init()**, p. 231.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **getenv()**, p. 262

setjmp()

Synopsis

```
#include <setjmp.h>
int setjmp( jmp_buf env );
```

Library

cLib.a

Description

setjmp() copies the current execution context into *env* for potential restoration using **longjmp()**.

Note: **setjmp()** should be used only in simple expressions, such as the entire controlling expression of an if statement. If used in complex expressions, **setjmp()** cannot guarantee that the proper elements are overwritten upon reentry.

When **setjmp()** returns directly, it returns 0. When the context is restored using **longjmp()**, **setjmp()** returns the value specified by **longjmp()**, if it is not 0. If the **longjmp()** function specifies 0, **setjmp()** returns 1.

Errors

None.

Example

See **shm_open()**, p. 734.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.6.1
- **longjmp()**, p. 392

Synopsis

```
#include <locale.h>

char *setlocale( int category, const char *locale );
```

Library

cLib.a

Description

setlocale() queries all or portions of the program's locale as specified by *category* and the object pointed to by *locale*.

Because the OS Open implementation of **setlocale()** supports only the C locale, the locale is always set as if by the following call:

```
setlocale(LC_ALL, "C");
```

If successful, **setlocale()** returns the requested category information. Otherwise, **setlocale()** returns a NULL pointer.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.4.2.1

setprompt()

Synopsis

```
#include <shell.h>
int setprompt( char *string );
```

Library

shell.o

Description

setprompt() sets the OpenShell prompt to the string pointed to by *string*.

setprompt() returns -1 if the string is longer than 15 characters, not including trailing null characters ($\backslash 0$). Otherwise, **setprompt()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

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

int setsockopt( int s, int level, int optname, const void *optval, int optlen );
```

Library

tcpipLib.a

Description

setsockopt() sets options associated with the socket *s*.

Options may exist at multiple protocol levels; they are always present at the highest (socket) protocol level.

When manipulating socket options, the protocol level and name of the option must be specified. To manipulate options at the socket level, *level* is specified as **SOCKET**. To manipulate options at all other levels, *level* is specified as the protocol number of the appropriate protocol controlling option. For example, if an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP. See **getprotobyname()** on page 280.

The object pointed to by *optval* is set to a non-zero value to enable an option. To disable an option, set the value of the object pointed to by *optval* to 0. *optlen* contains the size of the buffer pointed to by *optval*. If no option is given, *optval* can be 0.

optname and any specified options are passed to the appropriate protocol module for interpretation. The file **<sys/socket.h>** defines the following socket-level options, which are recognized at the socket level. Except as noted, each may be examined using **getsockopt()** and set using **setsockopt()**.

SO_LINGER	Lingers on a close subroutine if data is present. This option controls the action taken when an unsent messages queue exists for a socket and a thread performs a close on the socket.
SO_DEBUG	To conserve memory, this option is disabled.
SO_KEEPALIVE	Keeps connections active. Enables or disables the periodic transmission of messages on a connected socket. If the connected socket fails to respond to these messages, the connection is broken.

setsockopt()

SO_DONTROUTE	Does not apply routing on outgoing messages. Indicates outgoing messages should bypass the standard routing facilities. Directs messages to the appropriate network interface according to the network portion of the destination address. This option enables or disables routing of outgoing messages.
SO_USELOOPBACK	Uses the loopback interface for transmissions.
SO_BROADCAST	Specifies whether transmission of broadcast messages is supported to enable or disable broadcast support.
SO_REUSEADDR	Specifies that the rules used in validating addresses supplied by a bind subroutine should allow reuse of local addresses. This option enables or disables reuse of local addresses.
SO_OOBINLINE	Leaves received out-of-band data (data marked urgent) in line. This option enables or disables the receipt of out-of-band data.
SO_SNDBUF	Sets send buffer size information.
SO_RCVBUF	Sets receive buffer size information.
SO_SNDLOWAT	Sets send low-water mark information.
SO_RCVLOWAT	Sets receive low-water mark information.
SO_SNDTIMEO	Sets send time-out information.
SO_RCVTIMEO	Sets receive time-out information.
SO_ERROR	Retrieves information about error status and clear.
SO_TYPE	Retrieves information about a socket type.

If successful, **setsockopt()** returns 0. Otherwise, **setsockopt()** returns `-1` and sets *errno*.

Errors

[EBADF]	The descriptor <i>s</i> is not valid.
[ENOTSOCK]	The descriptor <i>s</i> references a file, not a socket.
[ENOPROTOOPT]	The option is unknown at the level indicated.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.8
- **getprotobyname()**, p. 280
- **getsockopt()**, p. 296
- **socket()**, p. 760

setsysconf()

Synopsis

```
#include <sys/ksyscall.h>
int setsysconf( int setvar, long value );
```

Library

rtxLib.a

Description

setsysconf() allows an application to alter the values returned by **sysconf()**.

This facility is provided so that platform-specific software provided by application designers can set these fields to the appropriate values. Any value that can be retrieved using the constants in **<unistd.h>** can be set with this function. *setvar* specifies the constant to be set to *value*.

If successful, **setsysconf()** returns 0. Otherwise, **setsysconf()** returns -1 and sets *errno*.

Errors

[EINVAL] *setvar* is not valid.

Example

See **fs_init()**, p. 231.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **sysconf()**, p. 841

Synopsis

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

int setuname( int setvar, const char* name );
```

Library

rtxLib.a

Description

setuname() allows an application to alter selected fields in the **struct utsname** structure returned by **uname()**.

Platform-specific application software can set these fields to the appropriate values, which can then be specified for *setvar*.

SET_NODENAME	Set <i>utsname.nodename</i>
SET_MACHINE	Set <i>utsname.machine</i>

If the string pointed to by *name* is longer than **_UNAME_LENGTH**, it is truncated to this length without error.

Note: **setuname()** is implemented as an application program to be called only during system initialization. **setuname()** performs no serialization with **uname()** or itself. Attempts by multiple threads to call these functions simultaneously may lead to unpredictable results.

If successful, **setuname()** returns 0. Otherwise, **setuname()** returns -1 and sets *errno*.

Errors

[EINVAL]	<i>setvar</i> is not valid.
----------	-----------------------------

Example

See **shm_open()**, p. 734.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **uname()**, p. 920

setvbuf()

Synopsis

```
#include <stdio.h>

int setvbuf( FILE *stream,
             char *buf,
             int type,
             size_t size );
```

Library

fsLib.a

Description

setvbuf() controls buffering and buffer size for the stream pointed to by *stream*.

stream must refer to an open file that has not been read or written. The array that *buf* points to serves as a buffer. If *buf* is NULL, an OS Open default buffer is used.

If the stream is buffered, *type* is specified. *type* must be `_IONBF`, `_IOFBF`, or `_IOLBF`. If *type* is `_IOFBF` or `_IOLBF`, *size* specifies the buffer size. If *type* is `_IONBF`, the stream is unbuffered and *size* and *buf* are ignored. Interactive devices are line-buffered.

<code>_IONBF</code>	No buffer is used.
<code>_IOFBF</code>	Full buffering is used for input and output. You should use <i>buf</i> as the buffer and <i>size</i> as the size of the buffer.
<code>_IOLBF</code>	Line buffering is used. The buffer is flushed when a new-line character is written, when the buffer is full, or when input is requested. The value for <i>size</i> must be greater than 0.

setvbuf() returns nonzero if *type* or *size* is not valid. If successful, **setvbuf()** returns 0.

Note: The file system must have been initialized using **fs_init()**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.5.6
- **fclose()**, p. 190
- **fflush()**, p. 197
- **fopen()**, p. 210
- **fs_init()**, p. 231
- **setbuf()**, p. 720

shell()

Synopsis

```
#include <shell.h>
void *shell( void *arg );
```

Library

shell.o

Description

shell() is the main entry point for OpenShell.

The parameter *arg* is ignored.

Note: The file system should be initialized by **fs_init()** before starting OpenShell; the files **stdin**, **stdout**, and **stderr** must be open.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- **shell_threadid()**, p. 733

Synopsis

```
#include <pthread.h>
pthread_t shell_threadid();
```

Library

shell.o

Description

shell_threadid() returns the thread ID of the thread that is running OpenShell, if OpenShell is running.

If OpenShell is not running, **shell_threadid()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **shell()**, p. 732

shm_open()

Synopsis

```
#include <sys/mman.h>
#include <fcntl.h>

int shm_open( const char *name, int oflag, mode_t mode );
```

Library

devLib.a

Description

shm_open() establishes a connection between a shared memory object and a file descriptor.

shm_open() returns a non-negative integer representing the file descriptor of the shared memory object. The file descriptor is used by other functions to refer to the shared memory object.

Note: OS Open without virtual memory provides one address space implicitly shared by all software, so traditional shared memory objects have little utility.

For compatibility with POSIX application profiles, an OS Open application program can specify a shared memory object at a specific real address or dynamically create a named memory region. A fixed address could be used to define memory mapped I/O in a portable way.

With virtual memory, an OS Open application program can specify a shared memory object at a virtual address, but not at a real address.

In order to reclaim the memory allocated to the shared memory object, **close()** must be called after the shared memory object is removed using a *shm_unlink()* call. For each **shm_open()** call, **close()** must be called.

The string pointed to by *name*, which names a shared memory object, must contain fewer than **NAME_MAX** characters from the portable file name set. In OS Open, the name appears in the file system name space and must be prefixed by *"/shm/"*.

The file descriptor returned by **shm_open()** is the lowest available file descriptor.

The file status flags and file access modes are set according to the value of *oflag*. The flags are:

[O_RDONLY]	Open for read access only.
[O_RDWR]	Open for read/write access.

[O_CREAT]	If the shared memory object exists, this flag has no effect. Otherwise, the shared memory object is created with a size of 0. Use ftruncate() or ioctl(fd, SET_SHM) to determine the size or extent of the shared memory object. Newly allocated space is set to 0.
[O_EXCL]	If O_EXCL and O_CREAT are specified, shm_open() fails if the shared memory object exists.
[O_TRUNC]	If a shared memory object exists and is opened using O_RDWR , the object is truncated to zero length. When a shared memory object is created, the state of the object persists until the object is unlinked, using shm_unlink() , and all references to the object disappear.

This implementation does not use the *mode* parameter.

If **shm_open()** fails, it returns **-1** and sets *errno*.

Errors

[EEXIST]	O_CREAT and O_EXCL were specified for an existing shared memory object.
[EMFILE]	No free file descriptors are available; the number of open files exceeds the number specified when the device subsystem was initialized.
[ENAMETOOLONG]	The length of the path string exceeds PATH_MAX .
[ENOENT]	The named shared memory object does not exist and the O_CREAT flag was not specified.
[EINVAL]	O_RDWR and O_RDONLY are mutually exclusive.

Example

The following example creates a shared memory object with a size of 12 288 bytes, maps a section to store the current operating system information and a time stamp, and creates a semaphore to protect this information. Non-local jumps are used for error recovery.

```
#include <setjmp.h>
#include <sys/utsname.h>
#include <sys/ksyscall.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stddef.h>
#include <semaphor.h>
#include <time.h>

#define K3 12288
```

shm_open()

```
#define K1 4096

void sys_info(int, jmp_buf);
sem_t system_info_sem;

int shared_init(void)
{
    int fd, rv;
    jmp_buf env;
    /* Create shared memory object */
    fd = shm_open("/shm/memory", O_CREAT | O_EXCL | O_RDWR, 0);
    if (fd < 0)
        return(fd);
    /* Set the size to K3(12288) bytes */
    if (ftruncate(fd, K3) < 0)
    {
        shm_unlink("/shm/memory");
        close(fd);
        return(-1);
    }
    /* set for nonlocal jump in case in error occurs in sys_info() */
    rv = setjmp(env);
    if (rv == 0)
        sys_info(fd, env); /* call function to store system info */
    /* An error occurred in sys_info() */
    else
    {
        shm_unlink("/shm/memory");
        close(fd);
        return(-1);
    }
    return(fd);
}

void sys_info(int fd, jmp_buf env)
{
    struct utsname info;
    void *addr;
    time_t time_stamp;
    char buffer[26];
    /* Set node name and machine name */
    if (setuname(SET_MACHINE, "tardis") < 0)
        longjmp(env, 1);
    if (setuname(SET_NODENAME, "timelord") < 0)
        longjmp(env, 1);
    /* Get system information */
```



```
if (uname(&info) < 0)
    longjmp(env, 2);
/* map section of memory for system information */
addr = mmap(NULL, sizeof(info) + 26, PROT_WRITE, MAP_SHARED, fd,
K1);
if (addr == (void *)-1)
    longjmp(env, 3);
/* Create a global semaphore to guard this information */
if (sem_init(&system_info_sem, 0, 1) < 0)
{
    munmap(addr, sizeof(info) + 26);
    longjmp(env, 4);
}
/* Copy system information to the mapped memory */
memcpy(addr, (void *)&info, sizeof(info));
/* get the time stamp */
time_stamp = time(NULL);
ctime_r(&time_stamp, buffer);
addr = (char *)addr + sizeof(info);
memcpy(addr, (void *)buffer, 26);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

References

- **close()**, p. 127
- IEEE Draft Standard 1003.4/D13, §12.2.1
- **shm_unlink()**, p. 738

shm_unlink()

Synopsis

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

int shm_unlink( const char *name );
```

Library

devLib.a

Description

shm_unlink() removes the name of the shared memory object named by the string pointed to by *name*.

If one or more references exist, the name is removed when **shm_unlink()** returns, but the removal of the object is postponed until all references are removed.

If successful, **shm_unlink()** returns 0. Otherwise, **shm_unlink()** returns -1 and sets *errno*.

Errors

[ENAMETOOLONG]	The length of the path string exceeds PATH_MAX , which is defined in the file <limits.h> .
[ENOENT]	The named shared memory object does not exist.

Example

See **shm_open()**, p. 734.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4/D13, §12.3.2
- **shm_open()**, p. 734

Synopsis

```
include <sys/socket.h>
int shutdown( int s, int how );
```

Library

tcpipLib.a

Description

shutdown() shuts down all or part of a full-duplex connection on the socket *s*.

If *how* is 0, further receives are disallowed. If *how* is 1, subsequent sends are disallowed. If *how* is 2, further sends and receives are disallowed.

If successful, **shutdown()** returns 0. Otherwise, **shutdown()** returns -1 and sets *errno*.

Errors

[EBADF]	The descriptor <i>s</i> is not valid.
[ENOTSOCK]	The descriptor <i>s</i> references a file, not a socket.
[ENOTCONN]	The socket <i>s</i> is not connected.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.12
- **connect()**, p. 132
- **recvfrom()**, p. 643
- **sendto()**, p. 718
- **socket()**, p. 760

sigaddset()

Synopsis

```
#include <signal.h>
int sigaddset( sigset_t *set, int signo );
```

Library

rtxLib.a

Description

sigaddset() adds the signal *signo* to the signal set pointed to by *set*.

If successful, **sigaddset()** returns 0. Otherwise, **sigaddset()** returns `-1` and sets *errno*.

Errors

[EINVAL] *signo* is a non-valid or unsupported signal number.

Example

See **sigpending()**, p. 748.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, §3.3.3
- **pthread_sigmask()**, p. 594
- **sigdelset()**, p. 741

Synopsis

```
#include <signal.h>

int sigdelset( sigset_t *set, int signo );
```

Library

rtxLib.a

Description

sigdelset() deletes the signal *signo* from the signal set pointed to by *set*.

If successful, **sigdelset()** returns 0. Otherwise, **sigdelset()** returns -1 and sets *errno*.

Errors

[EINVAL] *signo* is an non-valid or unsupported signal number.

Example

See **sigwait()**, p. 751.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, §3.3.3
- **pthread_sigmask()**, p. 594
- **sigaddset()**, p. 740

sigemptyset()

Synopsis

```
#include <signal.h>
int sigemptyset( sigset_t *set );
```

Library

rtxLib.a

Description

sigemptyset() initializes the signal set pointed to by *set* to exclude all valid OS Open signals.

Applications should call either **sigemptyset()** or **sigfillset()** at least once to initialize the signal set before its use by any signal function.

The **sigemptyset()** function returns 0.

Errors

None.

Example

See **sigpending()**, p. 748.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, §3.3.3
- **pthread_sigmask()**, p. 594
- **sigaddset()**, p. 740

Synopsis

```
#include <signal.h>

int sigfillset( sigset_t *set );
```

Library

rtxLib.a

Description

sigfillset() initializes the signal set pointed to by *set* to include all valid OS Open signals.

Applications should call either **sigemptyset()** or **sigfillset()** at least once to initialize the signal set before its use by any other signal function.

The **sigfillset()** function returns 0.

Errors

None.

Example

See **sigwait()**, p. 751.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, §3.3.3
- **pthread_sigmask()**, p. 594
- **sigaddset()**, p. 740

sigismember()

Synopsis

```
#include <signal.h>

int sigismember( sigset_t *set, int signo );
```

Library

rtxLib.a

Description

sigismember() tests whether *signo* is a member of the signal set pointed to by *set*.

sigismember() returns 1 if *signo* is a member of the set or 0 if *signo* is not. Otherwise, **sigismember()** returns -1 and sets *errno*.

Errors

[EINVAL] *signo* is a non-valid or unsupported signal number.

Example

See **sigpending()**, p. 748.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, §3.3.3
- **pthread_sigmask()**, p. 594

Synopsis

```
#include <setjmp.h>

void siglongjmp( sigjmp_buf env, int val );
```

Library

rtxLib.a

Description

siglongjmp() restores the environment saved by a call to **sigsetjmp()**.

If the **env** context contains a saved signal mask (See **sigsetjmp()** on page 750), **siglongjmp()** restores the mask. **siglongjmp()** returns *val* to the original call to **sigsetjmp()**, as if it had just returned. If *val* is 0, **sigsetjmp()** returns 1.

siglongjmp() changes the context of execution and does not return to its caller.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §8.3.1
- **sigsetjmp()**, p. 750

signal()

Synopsis

```
#include <signal.h>
void ( *signal (int sig, void (*func) (int))) ( int );
```

Library

cLib.a

Description

signal() handles receipt of the signal number *sig*.

sig must be generated by **raise()**.

If *func* is **SIG_DFL**, default signal handling equivalent to the following call occurs:

```
pthread_kill(pthread_self(), sig)
```

At program startup, the equivalent of:

```
signal(sig, SIG_DFL);
```

is executed for all signals.

If *func* is **SIG_IGN**, the signal is ignored. Otherwise, *func* points to a signal handler function that is called when the signal is received.

If a signal occurs and *func* points to a signal handler, the equivalent of:

```
signal(sig, SIG_DFL);
```

is executed, followed by the equivalent of:

```
(*func) (sig);
```

If *func* terminates by executing a **return** statement, the calling program resumes execution at the point of interruption.

If successful, **signal()** returns the value of *func* for the most recent **signal()** call for *sig*. Otherwise, **signal()** returns **SIG_ERR** and stores a positive number in *errno*.

Errors

[EINVAL] *sig* is not a valid signal number.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.7.1.1
- **pthread_kill()**, p. 568
- **raise()**, p. 630

Synopsis

```
#include <kadtLib.h>
int signal_list( char **bufp );
```

Library

kadtLib.a

Description

signal_list() provides formatted information about system signals.

The information is part of the **kda_dump()** display.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

signal_list() returns 0.

Errors

None.

Example

The following shows a sample return from **signal_list()**.

```
OS OPEN>signal_list()
Signals pending
(none)
-----
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84
- **kda_dump()**, p. 349

sigpending()

Synopsis

```
#include <signal.h>

int sigpending( sigset_t *set );
```

Library

rtxLib.a

Description

sigpending() stores, in the location pointed to by *set*, the signals pending on the calling thread and OS Open.

sigpending() updates *set* and returns 0.

Errors

None.

Example

The following example checks to see if the signal **SIGUSR1** is currently pending. If it is, the function receives it.

```
#include <signal.h>

int sig;

void clear_sigusr1(void)
{
    sigset_t sig_set;
    /* get the set of current pending signals */
    sigpending(&sig_set);
    /* Check to see if SIGUSR1 signal is pending */
    if(sigismember(&sig_set, SIGUSR1))
    {
        /* Receive the signal */
        sigemptyset(&sig_set);
        sigaddset(&sig_set, SIGUSR1);
        sigwait(&sig_set, &sig);
    }
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §3.3.6
- **pthread_sigmask()**, p. 594

sigsetjmp()

Synopsis

```
#include <setjmp.h>

int sigsetjmp( sigjmp_buf env, int savemask );
```

Library

rtxLib.a

Description

sigsetjmp() copies the current execution context into **env** for potential restoration by **siglongjmp()**.

If *savemask* is not zero, the current signal mask of the thread is also saved into **env**.

When **sigsetjmp()** returns directly, it returns 0. When the context is restored via **siglongjmp()**, **sigsetjmp()** returns the value specified by **siglongjmp()**, if it is not 0. If **siglongjmp()** specifies 0, 1 is returned.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.4a/D8, §8.3.1
- **siglongjmp()**, p. 745

Synopsis

```
#include <signal.h>

int sigwait( const sigset_t *set, int *sig );
```

Library

rtxLib.a

Description

sigwait() waits for a signal that is in the signal set pointed to by *set*, atomically clears it from the set of pending signals, and stores its signal number in the location pointed to by *sig*.

If no signal in the set is pending, the calling thread is blocked until one or more signals become pending.

If multiple threads use **sigwait()** to wait for the same signal, one of these threads will return from **sigwait()** with the signal number. It is unspecified which thread is selected.

If successful, **sigwait()** returns 0. Otherwise, **sigwait()** returns an error number.

Errors

[EINVAL]	<i>set</i> contains an non-valid or unsupported signal number. <i>set</i> specifies an empty set.
----------	---

Example

The following example waits on all signals except **SIGUSR1** and **SIGUSR2**. If the signal **SIGALRM** is received, **SIGUSR1** is sent to the thread. If the signal **SIGINT** is received, the **SIGUSR2** is sent to the thread. The function counts the number of **SIGINT** signals received. Otherwise, the function sleeps for two seconds before trying again.

```
#include <signal.h>
#include <pthread.h>

int sigint_count = 0;
int sig;

void *wait(void *arg)
{
    int rv=0;
    sigset_t sig_set;
    pthread_t thread_id = (pthread_t)arg;
    /* Set signal set to wait on all threads but SIGUSR1 and SIGUSR2 */
    sigfillset(&sig_set);
    sigdelset(&sig_set, SIGUSR1);
```

sigwait()

```
sigdelset(&sig_set, SIGUSR2);
/* Wait on the signals */
while(rv == 0)
{
    rv = sigwait(&sig_set, &sig);
    if(sig == SIGALRM)
        pthread_kill(thread_id, SIGUSR1);
    else if(sig == SIGINT)
    {
        pthread_kill(thread_id, SIGUSR2);
        sigint_count++;
    }
    else
        sleep(2);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4a/D8, §3.3.10

Synopsis

```
#include <math.h>
double sin( double x );
```

Library

mathLib.a

Description

sin() returns the sine of *x* (measured in radians).

If the result is out of range, **sin()** sets *errno*.

Errors

[ERANGE]	The result is out of range.
----------	-----------------------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.2.6

sinh()

Synopsis

```
#include <math.h>
double sinh( double x );
```

Library

mathLib.a

Description

sinh() returns the hyperbolic sine of *x*.

If the magnitude of *x* is too large, a range error occurs and **sinh()** sets *errno*.

Errors

[ERANGE]	The magnitude of <i>x</i> is too large.
----------	---

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.3.2

Synopsis

```
#include <unistd.h>

unsigned int sleep( unsigned int seconds );
```

Library

rtxLib.a

Description

sleep() causes the current thread to suspend until the number of real time seconds specified by *seconds* elapses or an unmasked signal is delivered to the calling thread.

If **sleep()** returns because the requested time has elapsed, 0 is returned. If **sleep()** returns due to the delivery of a signal, the value returned is the remaining time ((requested time) – (time suspended)) in seconds.

Errors

None.

Example

See **sigwait()**, p. 751.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §3.4.3
- **alarm()**, p. 58

slip_attach()

Synopsis

```
#include <netinet/if_slvar.h>
int slip_attach( int fd );
```

Library

netLib.a

Description

slip_attach() connects a serial interface to TCP/IP protocol stacks.

fd represents the file descriptor that will be used for reading and writing, and the file descriptor should be open so that reading and writing to it is possible.

XON/XOFF flow control must be disabled for the file descriptor that will be used by the serial line Internet Protocol (SLIP) interface. Only one SLIP interface can be used at a time.

Note: A maximum of eight SLIP interfaces are concurrently supported in OS Open.

If successful, **slip_attach()** returns 0. Otherwise, **slip_attach()** returns -1.

Errors

None.

Example

The following example sets up a serial interface for use with TCP/IP.

```
int devhandle;
int fd;

:

if (driver_install(&devhandle, async_init)) {
    (void)printf("driver_install() failed\n");
    return(-1);
}
if (device_install("/dev/s0", CHRTYPE, devhandle, 2, 1024, 1024)) {
    (void)printf("device_install() failed\n");
    return(-1);
}
fd=open("/dev/s0", O_RDWR, asyncParityNone, asyncParityOdd,
        asyncStopBits1, asyncDataBits8, 9600);
(void)ioctl(fd, ASYNCXONDISABLE);
if (tcpip_init("myhost", 1, 200)) {
    (void)printf("tcpip_init() failed\n");
    return(-1);
}
```

```
}
if (net_init()) {
    (void)printf("net_init() failed\n");
    return(-1);
}
if (slip_attach(fd)) {
    (void)printf("slip_attach() failed\n");
    return(-1);
}
if (ifconfig("sl0 myhost tanis netmask 255.255.240.0")) {
    (void)printf("ifconfig() failed\n");
    return(-1);
}
(void)ifconfig("sl0 up");
(void)route("add default tanis 1");
(void)ping("brightblade 4096 10");
```

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- `ifconfig()`, p. 304
- `slip_resume()`, p. 758
- `slip_suspend()`, p. 759

slip_resume()

Synopsis

```
#include <netinet/if_slvar.h>

int slip_resume( int instance, int fd )
```

Library

netLib.a

Description

slip_resume() resumes execution of the SLIP receive thread that was suspended using **slip_suspend()**.

instance specifies the instance number of the SLIP interface. For example, SLIP interface “sl0” has instance number 0 and SLIP interface “sl5” has instance number 5. *fd* specifies the file descriptor to be used for reading and writing. The file descriptor should be open for reading and writing.

After resuming a SLIP receive thread, a SLIP interface can be activated using the **ifconfig()** function.

If successful, **slip_resume()** returns 0. Otherwise, **slip_resume()** returns -1.

Errors

None.

Example

The following example brings up the SLIP interface “sl0”.

```
if(slip_resume(0,5)) {
    return(-1)
}
ifconfig("sl0 up")
```

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **ifconfig()**, p. 304
- **slip_attach()**, p. 756
- **slip_suspend()**, p. 759

Synopsis

```
#include <netinet/if_slvar.h>

int slip_suspend( int instance )
```

Library

netLib.a

Description

slip_suspend() suspends execution of a SLIP receive thread.

instance specifies the instance number of the SLIP interface. For example, SLIP interface “sl0” has instance number 0 and SLIP interface “sl5” has instance number 5.

Before the SLIP receive thread is suspended, the SLIP interface should be disabled by calling the **ifconfig()** function.

Note: This function can be used when a port or device used by a SLIP connection must be closed temporarily, such as when SLIP is run using a modem.

If successful, **slip_suspend()** returns 0. Otherwise, **slip_suspend()** returns -1.

Errors

None.

Example

Brings down SLIP interface “sl0”.

```
ifconfig("sl0 down")
if(slip_suspend(0)) {
    return(-1)
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **ifconfig()**, p. 304
- **slip_attach()**, p. 756
- **slip_resume()**, p. 758

socket()

Synopsis

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>

int socket( int domain, int type, int protocol );
```

Library

tcpipLib.a

Description

socket() creates an end point for communication and returns a descriptor.

socket() creates a socket in the specified *domain* of the specified *type*. *protocol* can be specified or assigned by OS Open. If *protocol* is not specified (a value of 0), OS Open selects an appropriate protocol from those protocols in the domain that can be used to support the requested socket type.

socket() returns a descriptor (an integer) that can be used in later subroutines that operate on sockets.

Domain can be assigned only **AF_INET** value, and *type* can be assigned the following values:

SOCK_STREAM	Provides sequenced, two-way byte streams with a transmission mechanism for out-of-band data.
SOCK_DGRAM	Provides datagrams, which are connectionless messages of a fixed maximum length (usually short).
SOCK_RAW	Provides access to internal network protocols and interfaces.

If successful, **socket()** returns an integer (the socket descriptor). Otherwise, **socket()** returns `-1` and sets *errno*.

Errors

[EMFILE]	The OS Open descriptor table is full.
[ENFILE]	The OS Open file table is full.
[ENOBUFS]	Insufficient resources were available in OS Open to complete the call.
[EPROTONOSUPPORT]	The protocol is not supported for the given domain.
[EPROTOTYPE]	The protocol type is not supported for the given domain.

Example

See **inet_aton()**, p. 309.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.12/D1, §7.5.13
- **bind()**, p. 80
- **connect()**, p. 132
- **recvfrom()**, p. 643
- **sendto()**, p. 718

socket_services()

Synopsis

```
#include <sys/ssLib.h>
int socket_services( int function, ... );
```

Library

ssLib.a

Description

socket_services() invokes PCMCIA Socket Services library functions.

The function number specified by *function* is passed to **socket_services()** along with parameters used by the specified PCMCIA Socket Services function.

<ssLib.h> contains PCMCIA Socket Services function numbers and PCMCIA Socket Services type definitions and constants.

Chapter 11, “OS Open PCMCIA Support,” in *OS Open User's Guide*, describes the OS Open binding of Socket Services functions. *PCMCIA Socket Services Specification* provides complete descriptions of the PCMCIA Socket Services functions.

Before **socket_services()** is called, **ss_init()** must be called to initialize the PCMCIA Socket Services library.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **ss_init()**, p. 766
- Chapter 11, “OS Open PCMCIA Support,” in *OS Open User's Guide*
- *PCMCIA Socket Services Specification*

Synopsis

```
#include <stdio.h>

int sprintf( char *buffer,
             const char *format-string, ... );
```

Library

cLib.a

Description

sprintf() formats and stores a series of characters and values in *buffer*, which is an array (unlike **printf()**, which writes to a stream).

Any argument list is converted and put out according to the corresponding format specification in *format-string*. *format-string* consists of ordinary characters and has the same form and function as *format-string* for **printf()**. See **printf()** on page 500 for a description of *format-string* and arguments.

sprintf() returns the number of characters written in the array, not counting the end NULL character.

Errors

None.

Example

See **read()**, p. 633.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.9.6.5
- **fprintf()**, p. 221
- **printf()**, p. 500
- **sscanf()**, p. 768

sqrt()

Synopsis

```
#include <math.h>
double sqrt( double x );
```

Library

mathLib.a

Description

sqrt() returns the non-negative square root of x .

If x is negative, a domain error occurs and **sqrt()** sets *errno*.

Errors

[EDOM]	x is negative.
--------	------------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.5.2

Synopsis

```
#include <stdlib.h>

void srand( unsigned int seed );
```

Library

cLib.a

Description

srand() sets the seed for producing a series of pseudo-random integers.

If **srand()** is not called, the **rand()** seed defaults as if **srand(1)** were called at program start. Any other value for *seed* sets the generator's seed to a different value.

rand() generates the pseudo-random numbers.

There is no return value.

Errors

None.

Example

See **fread()**, p. 225.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.2.2
- **rand()**, p. 631

ss_init()

Synopsis

```
#include <sys/ssLib.h>
int ss_init( int adapter_count, ... );
```

Library

ssLib.a

Description

ss_init() initializes the PCMCIA Socket Services library and the PCMCIA socket adapter hardware. After initialization, **socket_services()** can be called to invoke PCMCIA Socket Services functions.

ss_init() is passed at least four parameters. The first parameter, *adapter_count*, specifies the number of PCMCIA socket adapters controlled by the PCMCIA Socket Services handler.

Three positional parameters must be supplied for each adapter: its base system address and device index, and the number of sockets that the adapter controls. The device index is a byte value of 0x00 or 0x80, depending on the initial level of the SPKR_OUT/C_SEL pin of the PD6710 or PD672X chip.

Because **ss_init()** can be called only once, *adapter_count* must include all adapters controlled by the PCMCIA Socket Services handler, and the parameters for all adapters must be provided. Subsequent calls return -1 and set *errno*.

If successful, **ss_init()** returns 0.

Example

The following example initializes the PCMCIA Socket Services library for two adapters.

Both adapters share the base address 0x800003e0. The device index of the first adapter, which controls one socket, is 0x00. The device index of the second adapter, which controls two sockets, is 0x80. Note that the device index lets multiple adapters share a base address.

```
rc = ss_init(2, 0x800003e0, 0x00, 1, 0x800003e0, 0x80, 2);
```

Errors

[EINVAL]	Invalid input or ss_init() was called more than once.
[EIO]	Error communicating with socket adapter.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **socket_services()**, p. 762
- Chapter 11, “OS Open PCMCIA Support,” in *OS Open User's Guide*

sscanf()

Synopsis

```
#include <stdio.h>

int sscanf( const char *buffer,
            const char *format, ... );
```

Library

cLib.a

Description

sscanf() reads data from *buffer* into the locations given by the argument list.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in *format-string*. See **scanf()** on page 681 for a description of *format-string* and arguments.

sscanf() returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at end-of-string.

Errors

None.

Example

The following example reads various data from a string and then displays it on the standard output.

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

void read_string(char *tokenstring)
{
    int i;
    char s[80];
    char c;
    unsigned int h;
    memset((void *)s, '\0', 80);
    /* Read the data from the string */
    sscanf(tokenstring, "%s %i %c %x", s, &i, &c, &h);
    /* print data to stdout */
```



```
printf("string = %s\n", s);  
printf("decimal number = %d\n", i);  
printf("character = %c\n", c);  
printf("hexadecimal number = %x\n", h);  
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.9.6.6
- **fscanf()**, p. 234
- **printf()**, p. 500
- **scanf()**, p. 681

stat()

Synopsis

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

int stat( const char *path, struct stat *buf );
```

Library

devLib.a

Description

stat() stores information about the file pointed to by *path* in the object pointed to by *buf*.

If successful, **stat()** returns 0. Otherwise, **stat()** returns -1 and sets *errno*.

Errors

[EACCES]	Search permission is denied for a component of the path.
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of the path exceeds {PATH_MAX} .
[ENOENT]	<i>path</i> points to a file that does not exist or points to an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.

Example

See **access()**, p. 55.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.6.2
- See **fstat()**, p. 238.

Synopsis

```
#include <symLib.h>

unsigned long statsym_find_addr( unsigned long addr, char *symn );
```

Library

symLib.a

Description

statsym_find_addr() searches the static symbol table for the symbol with the address closest to *addr*.

The symbol name is copied to the character buffer pointed to by *symn* and the offset from the start of the symbol is returned. The buffer should be at least **SYMNMAX** characters long.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

statsym_find_ext()

Synopsis

```
#include <symLib.h>

unsigned long statsym_find_ext( char *symn );
```

Library

symLib.a

Description

statsym_find_ext() searches the static symbol table for an external symbol of name *symn*.

If successful, **statsym_find_ext()** returns the value of the symbol. Otherwise, **statsym_find_ext()** returns 0.

Errors

None.

Example

The following example provides the value of a symbol.

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

/* Display symbol value */
int sym_disp(char *symbol) {
    unsigned long symval;

    symval = statsym_find_ext(symbol);
    if (symval == 0) /* Symbol not found */
        return -1;
    printf("Symbol %s is %#x\n",symbol,symval);
    return 0;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

Synopsis

```
#include <symLib.h>

unsigned long statsym_find_ldsym( char *symn, ldrInfo_t *info );
```

Library

symLib.a

Description

statsym_find_ldsym() searches the static symbol table of a single loaded object for an external symbol of name *symn*.

info points to the **ldrInfo_t** structure returned from **ldrLoad()** and identifies which loaded object's symbol table to search.

If successful, **statsym_find_ldsym()** returns the value of the symbol. Otherwise, **statsym_find_ldsym()** returns 0.

Errors

None.

Example

The following example loads an object and provides the value of a symbol in that object.

```
#include <ldrLib.h>
#include <symLib.h>
#include <stdio.h>

/* Load an object, display symbol value then remove symbols */
int load_and_disp(int load_fd, char *ldname, char *symbol) {
    ldrInfo_t li;
    unsigned long symval;

    if (ldrQuery(load_fd, &li) != 0)
        return -1;
    if (ldrLoad(load_fd, ldname, &li) < 0)
        return -1;
    if (statsym_update(&li) != 0)
        return -1;
    symval = statsym_find_ldsym(symbol, &li);
    if (symval == 0) /* Symbol not found */
```

statsym_find_ldsym()

```
        return -1;
    printf("Symbol %s is %#x\n",symbol,symval);
    statsym_remove(&li);
    return 0;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **ldrLoad()**, p. 363

Synopsis

```
#include <symLib.h>
```

```
unsigned long statsym_find_name( char *symn, int *nfnd, char *file );
```

Library

symLib.a

Description

statsym_find_name() searches the static symbol table for the symbol of the name *symn*.

If *nfnd* is not NULL, the integer it points to is set to the number of symbols found. If *file* is not NULL, the file name where the symbol is defined is copied to the character buffer to which it points.

Both the symbol and file name buffers should be at least 40 characters long.

Note: **statsym_find_name()** does not find symbol names of function descriptors or TOCs; **statsym_find_toc()** should be called to find function descriptor symbols.

If successful, **statsym_find_name()** returns the address of the symbol. Otherwise, **statsym_find_name()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

statsym_find_tgaddr()

Synopsis

```
#include <symLib.h>

unsigned long statsym_find_tgaddr( unsigned long addr, char *symn,
                                   tg_t tg);
```

Library

symLib.a

Description

statsym_find_tgaddr() searches the static symbol table for the symbol with the address closest to *addr* within the scope of symbols available to the thread group *tg*.

The symbol name is copied to the character buffer pointed to by *symn* and the offset from the start of the symbol is returned. The buffer should be at least **SYMNMAX** characters long.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	No

Synopsis

```
#include <symLib.h>

unsigned long statsym_find_tgname( char *symn, int *nfnd, char *file,
                                tg_t tg );
```

Library

symLib.a

Description

statsym_find_tgname() searches the static symbol table for the symbol of the name *symn* within the scope of symbols available to the thread group *tg*.

If *nfnd* is not NULL, the integer it points to is set to the number of symbols found. If *file* is not NULL, the file name where the symbol is defined is copied to the character buffer to which it points.

The symbol and file name buffers should be at least **SYMNMAX** characters long.

Note: **statsym_find_tgname()** does not find symbol names of function descriptors or TOCs; **statsym_find_tgtoc()** should be called to find function descriptor symbols.

If successful, **statsym_find_tgname()** returns the address of the symbol. Otherwise, **statsym_find_tgname()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	No

statsym_find_tgtoc()

Synopsis

```
#include <symLib.h>

unsigned long statsym_find_tgtoc( char *symn, tg_t tg )
```

Library

symLib.a

Description

statsym_find_tgtoc() returns the address of the function descriptor or TOC of the symbol name pointed to by *symn*, if found. The scope of the names searched are symbols available to the thread group *tg*.

If unsuccessful, **statsym_find_tgtoc()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	No

Synopsis

```
#include <symLib.h>

unsigned long statsym_find_toc( char *symn )
```

Library

symLib.a

Description

statsym_find_toc() returns the address of the function descriptor or TOC of the symbol name pointed to by *symn*, if found.

Otherwise, **statsym_find_toc()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

statsym_init()

Synopsis

```
#include <symLib.h>
int statsym_init( int flags )
```

Library

symLib.a

Description

statsym_init() initializes internal tables required by the static symbol table functions.

statsym_init() must be called before calling any of the static symbol functions (those with names beginning with “statsym”).

The flags parameter decreases the amount of memory and function that the symbol table uses and provides. The following flags are valid:

STATSYM_INIT_ALL	Uses the most memory, fully functional.
STATSYM_INIT_NO_HID_EXT	Uses less memory than above, but cannot locate hidden or static functions.
STATSYM_INIT_NO_BY_ADDR	Uses less memory than STATSYM_INIT_ALL, but cannot find symbol name given an address.
STATSYM_INIT_NO_HID_EXT STATSYM_INIT_NO_BY_ADDR	Uses least amount of memory, but cannot locate hidden or static functions or symbol name given an address.

If successful, **statsym_init()** returns 0. Otherwise, **statsym_init()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <symLib.h>

unsigned long statsym_remove( ldrInfo_t *info );
```

Library

symLib.a

Description

statsym_remove() removes symbols from the symbol table added using **statsym_update()**.

If *info* is NULL, all symbols are removed, including the base-load symbols. Otherwise, *info* is the same argument used in the corresponding **statsym_update()**.

If successful, **statsym_remove()** returns 0. Otherwise, **statsym_remove()** returns -1.

Errors

None.

Example

See **statsym_find_ldsym()**, p. 773.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **ldrLoad()**, p. 363
- **statsym_update()**, p. 782

statsym_update()

Synopsis

```
#include <symLib.h>
unsigned long statsym_update( ldrInfo_t *info );
```

Library

symLib.a

Description

statsym_update() adds the symbols from an object loaded with **ldrLoad()** into the static symbol table.

info is the same argument used in **ldr_Load()**.

If successful, **statsym_update()** returns 0. Otherwise, **statsym_update()** returns -1.

Errors

None.

Example

See **statsym_find_ldsym()**, p. 773.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **ldrLoad()**, p. 363
- **statsym_remove()**, p. 781

Synopsis

```
#include <sys/devLib.h>
typedef struct block_req
{
    int request_type;    /* read or write */
    int block_id;        /* relative block number */
    void *block_buffer; /* block to read or write */
    int block_count;     /* number of blocks to transfer */
} block_req_t;

#define BLOCK_READ    0
#define BLOCK_WRITE   1
#define BLOCK_ERASE   2

int strategy( int filedес, block_req_t blkqrst );
```

Library

devLib.a

Description

strategy() is the interface for application users of block special files to request blocks of data to be written or read.

A block special file is organized as a linear list of blocks of a fixed size. To use this function, *request_type* is set to indicate whether the desired blocks should be read, written, or erased. *block_id* is set to the first relative block number for this request. *block_count* indicates the number of consecutive blocks to transfer, and *block_buffer* points to the storage area to be written from or read into.

If successful, **strategy()** returns 0. Otherwise, **strategy()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>filedes</i> does not refer to a valid file descriptor for a block special file.
[EINVAL]	The <i>request_type</i> is not valid.

Example

The following example reads data from a disk unit.

```
#include <sys/devLib.h>
#include <stdio.h>

int disk_read(int device_fd, int sector, int sect_count, char *addr)
{
    block_req_t blk_io;
    int rc;
```

strategy()

```
blk_io.request_type = BLOCK_READ;

blk_io.block_id = sector;
blk_io.block_buffer = addr;
blk_io.block_count = sect_count;
rc = strategy(device_fd, &blk_io);
return(rc);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **open()**, p. 478

Synopsis

```
#include <string.h>

int *strcasecmp( const char *s1, const char *s2 );
```

Library

cLib.a

Description

strcasecmp() compares the string pointed to by *s1* to the string pointed to by *s2*. This comparison ignores differences between upper-case and lower-case letters.

strcasecmp() returns an integer greater than, equal to, or less than zero, accordingly, as the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **memcmp()**, p. 403
- **strcmp()**, p. 790
- **strncmp()**, p. 800
- **strncasecmp()**, p. 798

strcat()

Synopsis

```
#include <string.h>

char *strcat( char *original, const char *append );
```

Library

cLib.a

Description

strcat() appends a copy of the string pointed to by *append* (including the terminating NULL character) to the end of the string pointed to by *original*.

The initial character of *append* overwrites the NULL character at the end of *original*. If copying takes place between objects that overlap, the behavior is undefined.

strcat() returns the value of *original*.

Errors

None.

Example

See **va_arg()**, p. 926.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.3.1
- **strncat()**, p. 799

Synopsis

```
#include <string.h>

char *strchr( const char *string, int c );
```

Library

cLib.a

Description

strchr() locates the first occurrence of *c* (converted to a *char*) in the string pointed to by *string*.

The terminating NULL character is considered to be part of the string.

strchr() returns a pointer to the located character or a NULL pointer if the character does not occur in the string.

Errors

None.

Example

The following example reads a file of names and changes their format from:

first_name [middle_initial] last_name

to the following format:

last_name, first_name [middle_initial]

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

#define NNames 512
#define MAXLEN 60

main(int argc, char *argv[])
{
    char *array[NNames];
    char gname[MAXLEN], lname[MAXLEN];
    FILE *fp;
    int count, num;
    char *name, string[MAXLEN], *cptr, *eptr;
    unsigned glength, length;
    /* Check for the number of arguments */
    if(--argc != 1)
    {
```

strchr()

```
    fprintf(stderr, "Usage: example filename\n");
    abort();
}
/* Open the file */
if((fp = fopen(argv[1], "r")) == NULL)
{
    fprintf(stderr, "Can not open %s\n", argv[1]);
    abort();
}
count = 0;
/* get line and examine it */
while(fgets(string, MAXLEN, fp) != NULL)
{
    /* find first character of the last name */
    if((cptr = strchr(string, '.')) != NULL)
        cptr += 2;
    else if((cptr = strchr(string, ' ')) != NULL)
        cptr++;
    else
        continue;
    /* copy last name to a buffer */
    strcpy(lname, cptr);
    /* change the new line character to a comma and add a space */
    eptr = strchr(lname, '\n');
    *eptr = ',';
    strcat(lname, " ");
    /* Get length of first name and possible middle initial */
    glength = (unsigned)(strlen(string) - strlen(cptr));
    /* copy first name and possible middle initial to a buffer */
    strncpy(gname, string, glength);
    /* concatenate the first name and middle initial to last name */
    name = strncat(lname, gname, glength);
    /* get length of entire name in new format */
    length = (unsigned)strlen(name);
    /* allocate storage for the new formatted name */
    array[count] = (char *)malloc(length + 1);
    strcpy(array[count], name);
    count++;
}
/* print out newly formatted names */
for(num = 0; num < count; num++)
    printf("%s\n", array[num]);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.5.2
- **memchr()**, p. 402
- **strchr()**, p. 803

strcmp()

Synopsis

```
#include <string.h>

int *strcmp( const char *s1, const char *s2 );
```

Library

cLib.a

Description

strcmp() compares the string pointed to by *s1* to the string pointed to by *s2*.

strcmp() returns an integer greater than, equal to, or less than zero, accordingly, as the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*.

Errors

None.

Example

See **getchar_unlocked()**, p. 258.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **memcmp()**, p. 403
- **strcasecmp()**, p. 785
- **strncmp()**, p. 800
- **strncasecmp()**, p. 798

Synopsis

```
#include <string.h>

int strcoll( const char *s1 const char *s2 );
```

Library

None. This is a macro defined in **<string.h>**.

Description

strcoll() calls **strcmp()**.

Because the OS Open implementation supports only the C locale, only the **LC_COLLATE** category of the C locale is available.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.4.3
- **setlocale()**, p. 723
- **strcmp()**, p. 790

strcpy()

Synopsis

```
#include <string.h>

char *strcpy( char *copy_to, const char *original );
```

Library

cLib.a

Description

strcpy() copies the string pointed to by *original* (including the terminating NULL character) into the array pointed to by *copy_to*.

If copying takes place between objects that overlap, the behavior is undefined.

strcpy() returns the value of *copy_to*.

Errors

None.

Example

See **strchr()**, p. 787.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.2.3
- **memcpy()**, p. 404
- **mempool_alloc()**, p. 417
- **strncpy()**, p. 801

Synopsis

```
#include <string.h>

size_t strcspn( const char *string, const char *characters );
```

Library

cLib.a

Description

strcspn() returns the length of the maximum initial segment of the string pointed to by *string* that consists entirely of characters not found in the string pointed to by *characters*.

strcspn() returns the length of the segment.

Errors

None.

Example

The following example verifies that there are no decimal digits in the string passed to it.

```
#include <string.h>

int verify_no_numbers(char *name)
{
    static char *decimal_numbers = {"0123456789"};
    if(strcspn(name, decimal_numbers) == strlen(name))
        return(1);
    else
        return(0);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.5.3
- **strspn()**, p. 804

strerror()

Synopsis

```
#include <string.h>
char *strerror( int errnum );
```

Library

cLib.a

Description

strerror() maps the error number in *errnum* to an error message string and returns a pointer to the string.

Errors

None.

Example

See **vfprintf()**, p. 930.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.6.2
- **clearerr()**, p. 99
- **ferror()**, p. 196
- **perror()**, p. 488

Synopsis

```
#include <time.h>

size_t strftime( char *dest,
                size_t maxsize,
                const char *format,
                const struct tm *timeptr );
```

Library

cLib.a

Description

strftime() places characters into the array pointed to by *dest* according to the string pointed to by *format*.

strftime() uses the C locale, which is the only locale supported in OS Open.

The format string contains a conversion specification characters array.

The characters that are converted and determined by the locale and by the values in the time structure pointed to by *timeptr*. The conversion specifiers and their meaning are listed below:

a	Insert the locale's abbreviated weekday name.
A	Insert the locale's full weekday name.
b	Insert the locale's abbreviated month name.
B	Insert the locale's full month name.
c	Insert the locale's date and time.
d	Insert the day of the month (01–31).
H	Insert the hour (24-hour clock) as a decimal number (00–23).
I	Insert the hour (12-hour clock) as a decimal number (01–12).
J	Insert the day of the year (001–366).
m	Insert the month (01–12).
M	Insert the minute (00–59).
p	Insert the locale's equivalent of either AM or PM.
S	Insert the second (00–61).
U	Insert the week number of the year (00–53), where Sunday is the first day of the week.
w	Insert the weekday (0–6), where Sunday is 0.

strftime()

W	Insert the week number of the year (00–53), where Monday is the first day of the week.
x	Insert the locale's date representation.
X	Insert the locale's time representation.
y	Insert the year without the century (00–99).
Y	Insert the year.
Z	Insert the name of the time zone, or no characters if the time zone is not available.
%%	Insert %.

The maximum number of characters that can be copied into the array is specified by *maxsize*.

If successful, **strftime()** returns the number of characters (bytes) placed into the array, not including the terminating NULL character. Otherwise, **strftime()** returns 0.

Errors

None.

Example

See **scanf()**, p. 681.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.12.3.5
- **setlocale()**, p. 723

Synopsis

```
#include <string.h>

size_t strlen( const char *s );
```

Library

cLib.a

Description

strlen() returns the length of the string pointed to by *s*.

strlen() returns the number of characters that precede the terminating NULL character.

Errors

None.

Example

See **strtok()**, p. 808.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.6.3

strncasecmp()

Synopsis

```
#include <string.h>
```

```
int strncasecmp( const char *s1, const char *s2, size_t n );
```

Library

cLib.a

Description

strncasecmp() compares not more than *n* characters (characters that follow a NULL character are not compared) from the string pointed to by *s1* to the string pointed to by *s2*. This comparison ignores differences between upper-case and lower-case letters.

strncasecmp() returns an integer greater than, equal to, or less than zero, accordingly, as the possibly NULL-terminated string pointed to by *s1* is greater than, equal to, or less than the possibly NULL-terminated string pointed to by *s2*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **memcmp()**, p. 403
- **strcasecmp()**, p. 785
- **strcmp()**, p. 790
- **strncmp()**, p. 800

Synopsis

#include <string.h>

char *strncat(char *original, const char *append, size_t n);

Library

cLib.a

Description

strncat() appends not more than *n* characters (a NULL character and characters that follow it are not appended) from the array pointed to by *append* to the end of the string pointed to by *original*.

The initial character of *append* overwrites the NULL character at the end of *original*. A terminating NULL character is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

strncat() returns the value of *original*.

Errors

None.

Example

See **strchr()**, p. 787.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.3.2
- **strcat()**, p. 786

strncmp()

Synopsis

```
#include <string.h>

int strncmp( const char *s1, const char *s2, size_t n );
```

Library

cLib.a

Description

strncmp() compares not more than *n* characters (characters that follow a NULL character are not compared) from the string pointed to by *s1* to the string pointed to by *s2*.

strncmp() returns an integer greater than, equal to, or less than zero, accordingly, as the possibly NULL-terminated string pointed to by *s1* is greater than, equal to, or less than the possibly NULL-terminated string pointed to by *s2*.

Errors

None.

Example

See **getchar_unlocked()**, p. 258.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **memcmp()**, p. 403
- **strcasecmp()**, p. 785
- **strcmp()**, p. 790
- **strncasecmp()**, p. 798

Synopsis

#include <string.h>

char *strncpy(**char** *copy_to, **const char** *original, **size_t** n);

Library

cLib.a

Description

strncpy() copies not more than *n* characters (characters that follow a NULL character are not copied) from the array pointed to by *original* to the array pointed to by *copy_to*. If copying takes place between objects that overlap, the behavior is undefined.

If the array pointed to by *original* is a string that is shorter than *n* characters, NULL characters are appended to the copy in the array pointed to by *copy_to*, until *n* characters have been copied.

strncpy() returns the value of *copy_to*.

Errors

None.

Example

See **strchr()**, p. 787.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.2.4
- **memcpy()**, p. 404
- **memmove()**, p. 415
- **strcpy()**, p. 792

strpbrk()

Synopsis

```
#include <string.h>

char *strpbrk( const char *string, const char *delimiters );
```

Library

cLib.a

Description

strpbrk() locates the first occurrence in the string pointed to by *string* of any character found in the string pointed to by *delimiters*.

strpbrk() returns a pointer to the character or a NULL pointer if no character from *delimiters* occurs in *string*.

Errors

None.

Example

The following example returns a pointer to the first occurrence of a white space or punctuation in a string.

```
#include <string.h>
#include <stddef.h>

char *findseparators(char *string)
{
    char *seps = {" \\n\\t!\"'#$%&\\'()*+,-./:;<=>?[\\]\\{|}"};
    if(string == NULL)
        return(NULL);
    return(strpbrk(string, seps));
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.5.4

Synopsis

```
#include <string.h>

char *strrchr( const char *s, int c );
```

Library

cLib.a

Description

strrchr() locates the last occurrence of *c* (converted to a **char**) in the string pointed to by *s*.

The terminating NULL character is considered to be part of the string.

strrchr() returns a pointer to the character or a NULL pointer if *c* does not occur in the string.

Errors

None.

Example

The following example truncates a string at the last occurrence of a given character.

```
#include <string.h>
#include <stddef.h>

char *truncate(char *string, char endpoint)
{
    char *endchar;

    if(string != NULL && (endchar = strrchr(string, endpoint)) != NULL)
        *(++endchar) = '\0';
    return(string);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.5.5
- **memchr()**, p. 402
- **strchr()**, p. 787

strspn()

Synopsis

```
#include <string.h>

size_t strspn( const char *string, const char *characters );
```

Library

cLib.a

Description

strspn() returns the length of the maximum initial segment of the string pointed to by *string*, which consists entirely of characters from the string pointed to by *characters*.

strspn() returns the length of the segment.

Errors

None.

Example

The following example verifies a string that meets the POSIX portability name set.

```
#include <string.h>

int verify_name(char *name)
{
    char *name_set =
    {"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012345
    6789.-_"};
    if(strspn(name, name_set) == strlen(name))
        return(1);
    else
        return(0);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.5.6
- **strcspn()**, p. 793

Synopsis

```
#include <string.h>

char *strstr( const char *original, const char *search );
```

Library

cLib.a

Description

strstr() locates the first occurrence in the string pointed to by *original* of the sequence of characters (excluding the terminating NULL character) in the string pointed to by *search*.

strstr() returns a pointer to the located string or a NULL pointer if the string is not found. If *search* points to a string with zero length, the function returns *original*.

Errors

None.

Example

See **fopen()**, p. 210.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.5.7

strtod()

Synopsis

```
#include <stdlib.h>

double strtod( const char *nptr, char **endptr );
```

Library

cLib.a

Description

strtod() converts the string pointed to by *nptr* to **double** representation and returns the converted value.

strtod() first decomposes the input string into three parts: an initial, possibly empty, sequence of whitespace characters (as specified by **isspace()**); a subject sequence resembling a floating-point constant; and a final string of one or more unrecognized characters, including the terminating NULL of the input string.

strtod() then attempts to convert the subject sequence to a floating-point number and returns the result.

strtod() expects a subject sequence of the following form: an optional plus or minus sign, then a non-empty sequence of digits optionally containing a decimal-point character, then an optional exponent part.

The components of the exponent part are one of either “e” or “E” followed by a exponent represented as an optionally signed digit sequence.

If the subject sequence is of the expected form, the sequence is interpreted as a floating point constant as defined in ANSI X3.159-1989, §3.1.3.1, except that the decimal-point character is used in place of a period, and that if neither an exponent part or decimal-point character is present, a decimal point is assumed to follow the last character in the subject sequence.

If the subject sequence begins with a minus sign, the resulting floating-point number is negated.

If *endptr* is not NULL, it points to the object in which a pointer to the final string is stored.

If the subject sequence is not of the expected form or is empty, no conversion is performed and the value of *nptr* is stored in the object pointed to by *endptr*, unless *endptr* is NULL.

If no conversion could be performed, **strtod()** returns 0. If the converted value is outside the range of representable values, **strtod()** returns plus or minus **HUGE_VAL**, according to the sign of the value, and sets *errno*. If the converted value would cause an underflow, **strtod()** returns 0 and sets *errno*.

Note: **strtod()** should be called only by threads enabled for floating-point arithmetic.

Errors

[ERANGE]	The converted value is not in the range of -HUGE_VAL through +HUGE_VAL or would cause an underflow.
----------	---

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §3.1.3.1
- ANSI X3.159-1989, §4.10.1.4

strtok()

Synopsis

```
#include <string.h>

char *strtok( char *original, const char *delimiters );
```

Library

cLib.a

Description

strtok(), in a sequence of calls, decomposes the string pointed to by *original* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *delimiters*.

The first call in the sequence has *original* as its first argument, and is followed by calls with a NULL pointer as their first argument. The separator string pointed to by *delimiters* may be different from call to call.

The first call in the sequence searches the string pointed to by *original* for the first character that is not contained in the current separator string pointed to by *delimiters*. If no such character is found, then there are no tokens in the string pointed to by *original* and **strtok()** returns a NULL pointer. If such a character is found, it is the start of the first token.

strtok() then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by *original*, and subsequent searches for a token return a NULL pointer. If such a character is found, it is overwritten by a NULL character, which terminates the current token. The **strtok()** function saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a NULL pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

strtok() returns a pointer to the first character of a token or a NULL pointer if there is no token.

Errors

None.

Example

The following example counts the number of words and the number of characters (excluding whitespace and punctuation characters) in a string.

```
#include <string.h>
#include <stdio.h>
#include <stddef.h>

void count(char *string)
```



```
{
    char *delimiters = {"\\t\\n .,:;?!\""};
    char buffer[120];
    char *wp;
    int words = 0, characters = 0;

    /* copy string to a working buffer because strtok destructively */
    /* separates the string into tokens. */
    strcpy(buffer, string);

    /* Split into words and count the characters in each word */
    wp = strtok(buffer, delimiters);
    if(wp != NULL)
    {
        words++;
        characters += strlen(wp);
        while((wp = strtok(NULL, delimiters)) != NULL)
        {
            words++;
            characters += strlen(wp);
        }
        printf("# of words: %d, # of characters: %d\\n", words,
            characters);
    }
    else
    {
        printf("String can not be split into words\\n");
        printf("# of characters: %d\\n", strlen(buffer));
    }
}

main()
{
    char buf[120];
    while(1)
    {
        if(puts("Enter a string") == EOF)
            break;
        if(gets(buf) == NULL)
            break;
        count(buf);
    }
}
```

strtok()

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.5.8
- **strtok_r()**, p. 811

Synopsis

```
#include <string.h>

char *strtok_r( char *s,
                const char *sep,
                char **lasts );
```

Library

cLib.a

Description

strtok_r() is a reentrant version of **strtok()**.

strtok_r() considers the NULL-terminated string *s* as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *sep*. The argument *lasts* points to a user-provided pointer, which in turn points to the position within *s* at which scanning should begin.

In the first call to **strtok_r()**, *s* points to a NULL-terminated string, *sep* to a NULL-terminated string of separator characters and *lasts* to a char **. **strtok_r()** returns a pointer to the first character of the first token, writes a NULL character into *s* immediately following the returned token, and updates the pointer to which *lasts* points so that it points to the first character following the NULL written into *s*.

In subsequent calls, *s* must be a NULL pointer and *lasts* must be unchanged so that subsequent calls will move through the string *s*, returning successive tokens until no tokens remain. The separator string *sep* may be different from call to call. When no token remains in *s*, a NULL pointer is returned.

strtok_r() returns a pointer to the token found or a NULL pointer when no token is found.

Errors

None.

Example

The following example breaks a string into individual tokens and puts pointers to the tokens into an array. It then returns the number of tokens created. No more than a given maximum number of tokens are created.

```
#include <string.h>
#include <stddef.h>

int tokenize(char *cmd_string, char *tokenlist[],
            size_t maxtoken)
{
    int tokencount;
```

strtok_r()

```
char *lasts;
char *tokensep = {"\t\n ,,"};
char *thistoken;
if(cmd_string == NULL || !maxtoken)
    return(0);
thistoken = strtok_r(cmd_string, tokensep, &lasts);
for(tokencount = 0; tokencount < maxtoken && thistoken != NULL; )
{
    tokenlist[tokencount++] = thistoken;
    thistoken = strtok_r(NULL, tokensep, &lasts);
}
tokenlist[tokencount] = NULL;
return(tokencount);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4a/D8, §19.2.3.1
- **strtok()**, p. 808

Synopsis

```
#include <stdlib.h>

long int strtol( const char *nptr,
                 char **endptr,
                 int base );
```

Library

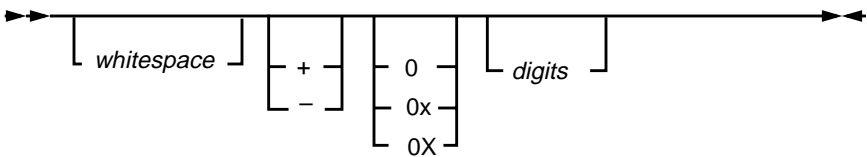
cLib.a

Description

strtol() converts a character string to a sequence of characters, each of which can be interpreted as a **long int**.

The function stops reading the string at the first character that is not recognized as part of a number. This character can be the NULL character at the end of the string. The ending character can also be the first numeric character greater than or equal to *base*.

strtol() expects *nptr* to point to a string with the following form:.



If *base* is from 2 through 36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16, or 10): the prefix 0 means base 8; the prefix 0x or 0X means base 16; using any other digit without a prefix means decimal.

strtol() returns the value represented in the string, except when the representation causes an overflow. If an overflow occurs, it returns **LONG_MAX** or **LONG_MIN**, according to the sign of the value.

endptr holds a pointer to the character the conversion stopped on. If the string pointed to by *nptr* does not have the expected form, no conversion is performed and the value of *nptr* is stored in the object pointed to by *endptr*.

Errors

[ERANGE]	An overflow occurred.
[EINVAL]	The specified base is not valid.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

strtol()

References

- ANSI X3.159-1989, §4.10.1.5
- **atoi()**, p. 72
- **atol()**, p. 74
- **strtoul()**, p. 815

Synopsis

```
#include <stdlib.h>

unsigned long int strtoul( const char *string1,
                          char **string2,
                          int base );
```

Library

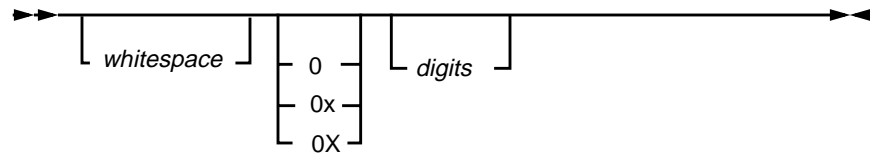
cLib.a

Description

strtoul() converts a character string to an unsigned long-integer value.

The input *string1* is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the string at the first character that it cannot recognize as part of the number. This character can be the first numeric character greater than or equal to *base*.

strtoul() sets *string2* to point to the character conversion stopped on. **strtoul()** expects *string1* to point to a string with the following form:



If *base* is from 2 through 36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16, or 10); the prefix 0 means base 8, the prefix 0x or 0X means base 16; using any other digit without a prefix means decimal.

strtoul() returns the value represented in the string or 0 if no conversion could be performed. If an overflow occurs, **strtoul()** returns *ULONG_MAX*.

Errors

[ERANGE]	An overflow occurred.
[EINVAL]	The specified base is not valid.

Example

See **flockfile()**, p. 205.

strtoul()

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.1.6
- **atoi()**, p. 72
- **atol()**, p. 74

Synopsis

```
#include <string.h>

int strxfrm( char *s1, const char *s2, size_t n );
```

Library

None. This is a macro defined in **<string.h>**.

Description

strxfrm() calls **strncpy()**.

Because the OS Open implementation supports only the C locale, only the **LC_COLLATE** category of the C locale is available. **strxfrm()** cannot affect string transformation.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.11.4.3
- **setlocale()**, p. 723
- **strcoll()**, p. 791
- **strcpy()**, p. 792

svc_destroy(), SVC_DESTROY()

Synopsis

```
#include <rpc/rpc.h>

void svc_destroy( SVCXPRT *xpvt );
```

Library

None. This is a macro defined in `<rpc/svc.h>`.

Description

svc_destroy(), also called **SVC_DESTROY()**, destroys the RPC service transport handle pointed to by *xprt*.

Destroying the service transport handle deallocates the private data structures, including the handle. After a **svc_destroy()** call, the handle is no longer defined.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **clnt_destroy()**, **CLNT_DESTROY()**, p. 107
- **rpc_thread_init()**, p. 676
- **svc_freeargs()**, **SVC_FREEARGS()**, p. 819

Synopsis

```
#include <rpc/rpc.h>

bool_t svc_freeargs( SVCXPRT *xpvt, xdrproc_t inproc, char *in );
```

Library

None. This is a macro defined in `<rpc/svc.h>`.

Description

svc_freeargs(), also called **SVC_FREEARGS()**, frees data allocated by RPC/XDR.

xpvt points to the RPC service transport handle. *inproc* specifies the XDR routine that decodes the arguments. *in* specifies the address where the allocated data is stored.

The freed data was allocated when RPC/XDR decodes arguments to a service procedure using **svc_getargs()**.

If successful, **svc_freeargs()** returns 1. Otherwise, **svc_freeargs()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **svc_destroy()**, **SVC_DESTROY()**, p. 818
- **svc_getargs()**, **SVC_GETARGS()**, p. 820

svc_getargs(), SVC_GETARGS()

Synopsis

```
#include <rpc/rpc.h>

bool_t svc_getargs( SVCXPRT *xprt, xdrproc_t inproc, char *in );
```

Library

None. This is a macro defined in `<rpc/svc.h>`.

Description

svc_getargs(), also called **SVC_GETARGS()**, decodes the arguments of an RPC request associated with the RPC service transport handle.

xprt points to the RPC service transport handle.

inproc specifies the XDR routine that decodes the arguments.

in specifies the address where the procedure arguments are placed.

If successful, **svc_getargs()** returns 1. Otherwise, **svc_getargs()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **svc_freeargs()**, **SVC_FREEARGS()**, p. 819

Synopsis

```
#include <netinet/in.h>
#include <rpc/rpc.h>
struct sockaddr_in *svc_getcaller( SVCXPRT *xprt );
```

Library

None. This is a macro defined in `<rpc/svc.h>`.

Description

svc_getcaller(), also called **SVC_GETCALLER()**, returns the network address of the caller of the procedure associated with the RPC service transport handle pointed to by *xprt*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **svc_register()**, p. 824
- **svc_run()**, p. 825

svc_getreq()

Synopsis

```
#include <rpc/svc.h>
void svc_getreq( int rdfs )
```

Library

rpcLib.a

Description

svc_getreq() processes incoming RPC requests.

rdfs is the read file descriptor mask returned by **select()**. **svc_getreq()** returns after processing all (up to 32) file descriptors specified by *rdfs*. If more file descriptors are required, **svc_getreqset()** can be used.

Note: **svc_getreqset()** is used only if the RPC server implements custom asynchronous event processing and does not use **svc_run()**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **select()**, p. 691
- **svc_getreqset()**, p. 823
- **svc_run()**, p. 825

Synopsis

```
#include <rpc/rpc.h>
void svc_getreqset( fd_set *rdfs );
```

Library

rpcLib.a

Description

svc_getreqset() is used only if a service implementer does not call **svc_run()**, but instead implements custom asynchronous event processing.

svc_getreqset(), which is called when **select()** determines that an RPC request arrived on any RPC sockets, returns after all sockets associated with the value specified by *rdfs* are serviced.

rdfs specifies the resultant read-file descriptor mask.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **select()**, p. 691
- **svc_run()**, p. 825

svc_register()

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t svc_register( SVCXPRT *xpvt, u_long prognum, u_long versnum,
                    void (*dispatch)(), int protocol );
```

Library

rpcLib.a

Description

svc_register() maps a remote procedure to the service dispatch procedure pointed to by *dispatch*.

If the data transport specified by *protocol* is 0, the service is not registered with the portmap daemon. If *protocol* is **IPPROTO_UDP** or **IPPROTO_TCP** (defined in the file **<netinet.h>**), the remote procedure triple (*prognum*, *versnum*, and *protocol*) is mapped to the **xpvt->xp_port** port.

xpvt points to an RPC service transport handle.

prognum specifies the program number of the remote program; *versnum* specifies its version number.

The dispatch procedure takes the following form:

```
dispatch(request, xpvt)
struct svc_req *request;
SVCXPRT *xpvt;
```

If successful, **svc_register()** returns 1. Otherwise, **svc_register()** returns a value if 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **pmap_getmaps()**, p. 491
- **pmap_set()**, p. 495
- **rpc_thread_init()**, p. 676
- **svc_unregister()**, p. 827

Synopsis

```
#include <rpc/rpc.h>
void svc_run();
```

Library

rpcLib.a

Description

svc_run() waits for an RPC service request to arrive.

When a request arrives, **svc_run()** calls the appropriate service procedure using **svc_getreqset()**. This service procedure is usually waiting for a **select()** to return.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **callrpc()**, p. 90
- **registerrpc()**, p. 649
- **rpc_thread_init()**, p. 676
- **select()**, p. 691
- **svc_getreqset()**, p. 823

svc_sendreply()

Synopsis

```
#include <rpc/rpc.h>
bool_t svc_sendreply( SVCXPRT *xpvt, xdrproc_t outproc, char *out );
```

Library

rpcLib.a

Description

svc_sendreply() returns the results of a remote procedure call.

xpvt points to an RPC service transport handle.

outproc specifies the XDR routine that encodes the results.

out points to the address where results are stored.

svc_sendreply() is called by a RPC service dispatch subroutine.

If successful, **svc_sendreply()** returns 1. Otherwise, **svc_sendreply()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

void svc_unregister( u_long prognum, u_long procnum );
```

Library

rpcLib.a

Description

svc_unregister() removes the mapping between a dispatch subroutine and a service procedure.

prognum specifies the program number of the service procedure; *versnum* specifies its version number.

svc_unregister() also removes the mapping between the port number and the service procedure

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **pmap_unset()**, p. 496
- **rpc_thread_init()**, p. 676
- **svc_register()**, p. 824

svcerr_auth()

Synopsis

```
#include <rpc/rpc.h>

void svcerr_auth( SVCXPRT *xpvt, enum auth_stat why );
```

Library

rpcLib.a

Description

svcerr_auth() is called by a service dispatch subroutine that cannot perform a remote procedure call because of a authentication error.

svcerr_auth() sets the status of the RPC reply message to **AUTH_ERROR**.

xpvt points to the RPC service transport handle.

why specifies the authentication error.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
void svcerr_decode( SVCXPRT *xprt );
```

Library

rpcLib.a

Description

svcerr_decode() is called by a service dispatch subroutine that cannot decode the parameters specified in a request.

svcerr_decode() sets the status of the RPC reply message to **GARBAGE_ARGS** condition.

xprt points to the RPC service transport handle.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **svc_getargs()**, **SVC_GETARGS()**, p. 820

svcerr_noproc()

Synopsis

```
#include <rpc/rpc.h>
void svcerr_noproc( SVCXPRT *xprt );
```

Library

rpcLib.a

Description

svcerr_noproc() is called by a service dispatch subroutine that cannot implement the procedure number the caller has requested.

svcerr_noproc() sets the status of the RPC reply message to **PROC_UNAVAIL**, indicating that the program cannot support the requested procedure.

Note: Service implementers do not usually need **svcerr_noproc()**.

xprt points to the RPC service transport handle.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
void svcerr_noprog( SVCXPRT *xprt );
```

Library

rpcLib.a

Description

svcerr_noprog() is called by a service dispatch subroutine when the requested program is not registered with the RPC package.

This subroutine sets the status of the RPC reply message to **PROG_UNAVAIL** condition, indicating that the remote server has not exported the program.

Note: Service implementers do not usually need **svcerr_noprog()**.

xprt points to the RPC service transport handle.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

svcerr_progvers()

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

void svcerr_progvers( SVCXPRT *xpvt, u_long low, u_long high );
```

Library

rpcLib.a

Description

svcerr_progvers() is called by a service dispatch subroutine when the requested version of a program is not registered with the RPC package.

svcerr_progvers() sets the status of the RPC reply message to **PROG_MISMATCH**, indicating that the remote server cannot support the client's version number.

Note: Service implementers do not usually need **svcerr_progvers()**.

xprt points to the RPC service transport handle.

low specifies the lowest supported version.

high specifies the highest supported version supported.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
void svcerr_systemerr( SVCXPRT *xprt );
```

Library

rpcLib.a

Description

svcerr_systemerr() is called by a service dispatch subroutine that detects a system error not covered by a protocol.

For example, a service dispatch subroutine calls **svcerr_systemerr()** if the first subroutine can no longer allocate storage. **svcerr_systemerr()** sets the status of the RPC replay message to the **SYSTEM_ERR** condition.

xprt points to the RPC service transport handle.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

svcerr_weakauth()

Synopsis

```
#include <rpc/rpc.h>
void svcerr_weakauth( SVCXPRT *xprt );
```

Library

rpcLib.a

Description

svcerr_weakauth() is called by a service dispatch subroutine that cannot make the remote call because the supplied authentication parameters are insufficient for security reasons.

svcerr_weakauth() calls **svcerr_auth()** with the correct RCP service transport handle pointed to by *xprt*.

svcerr_weakauth() sets the status of the RPC reply message to the **AUTH_TOOWEAK** condition as the authentication error **AUTH_ERR**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **svcerr_auth()**, p. 828
- **svcerr_decode()**, p. 829

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

SVCXPRT *svcfcreate( int sock, u_int sendsize, u_int recvsz );
```

Library

rpcLib.a

Description

svcfcreate() creates a service on any open socket.

sock identifies the socket.

sendsize specifies the size of the send buffer; *recvsz* specifies the size of the receive buffer.

If successful, **svcfcreate()** returns a TCP-based transport handle. Otherwise, **svcfcreate()** returns a NULL.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

svccraw_create()

Synopsis

```
#include <rpc/rpc.h>
SVCPRT *svccraw_create();
```

Library

rpcLib.a

Description

svccraw_create() creates an RPC service transport handle used for simulation.

The service transport handle is located within the address space of the process. The corresponding RPC server resides in the same address space, so simulation of RPC and acquisition of RPC overheads are performed.

If successful, **svccraw_create()** subroutine returns a pointer to a valid RPC transport handle. Otherwise, **svccraw_create()** returns NULL.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **clnt_raw_create()**, p. 116
- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
#include <types.h>

SVCXPRT *svctcp_create( int sock, u_int sendsize, u_int recvsize );
```

Library

rpcLib.a

Description

svctcp_create() creates an RPC service transport handle, using TCP/IP, and returns a pointer to the handle.

sock specifies the socket associated with the transport. If the value of *sock* is **RPC_ANYSOCK** (defined in the file **<rpc/svc.h>**), **svctcp_create()** creates a new socket.

The service transport handle socket number is set to **xprt->xp_sock**. If the socket is not bound to a local TCP/IP port, **svctcp_create()** binds the socket to an arbitrary port and sets its port number to **xprt->xp_port**.

If successful, **svctcp_create()** returns a valid RPC service transport handle. Otherwise, **svctcp_create()** returns a value of NULL.

Errors

[EADDINUSE]	The port is already in use
[EPFNOSUPPORT]	The protocol family is not supported.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **registerrpc()**, p. 649
- **rpc_thread_init()**, p. 676
- **svcudp_create()**, p. 839

svcudp_bufcreate()

Synopsis

```
#include <rpc/svc.h>
#include <types.h>
SVCXPRT *svcudp_bufcreate( int fd, u_int sendsz, u_int recvsz );
```

Library

rpcLib.a

Description

svcudp_bufcreate() creates an RPC service transport, using UDP/IP and supporting user-specified send and receive buffer sizes.

The service transport is associated with the file descriptor *fd*.

If *fd* is set to **RPC_ANYSOCK** (defined in the file **<rpc/svc.h>**), a new file descriptor is created. If the file descriptor is not bound to a local UDP port, **svcudp_bufcreate()** binds the file descriptor to an arbitrary port.

sendsz specifies the send buffer size; *recvsz* specifies the receive buffer size.

If successful, **svcudp_bufcreate()** returns a pointer to a RPC service transport. Otherwise, **svcudp_bufcreate()** returns a NULL.

Errors

[EADDINUSE]	The port is already in use
[EPFNOSUPPORT]	The protocol family is not supported.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **svccraw_create()**, p. 836
- **svctcp_create()**, p. 837
- **svcudp_create()**, p. 839

Synopsis

```
#include <rpc/rpc.h>
SVCPXPRT *svculdp_create( int sock );
```

Library

rpcLib.a

Description

svculdp_create() creates an RPC service transport handle using UDP/IP and returns a pointer to the handle.

RPC messages transported by UDP/IP can contain up to 8KB of encoded data; **svculdp_create()** is appropriate for programs that take arguments or return results smaller than 8KB.

sock specifies the socket associated with the transport handle. If the value of *sock* is **RPC_ANYSOCK** (defined in the file **<rpc/svc.h>**), **svculdp_create()** creates a new socket. and sets the service transport handle socket number to **xprt->xp_sock**.

If the socket is not bound to a local UDP/IP port, **svculdp_create()** binds the socket to an arbitrary port and sets the port number to **xprt->xp_port**.

If successful, **svculdp_create()** returns a valid RPC service transport handle. Otherwise, **svculdp_create()** returns a value of NULL.

Errors

[EADDINUSE]	The port is already in use
[EPFNOSUPPORT]	The protocol family is not supported.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **registerrpc()**, p. 649
- **rpc_thread_init()**, p. 676
- **svctcp_create()**, p. 837

symptr_get()

Synopsis

```
#include <kadtLib.h>

void *symptr_get( unsigned long *Nsyms );
```

Library

kadtLib.a

Description

symptr_get() returns the address of the static symbol table and sets the unsigned long variable pointed to by *Nsyms* to the number of symbols.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <unistd.h>
long sysconf( int name );
```

Library

rtxLib.a

Description

sysconf() allows a program to determine the value of a configurable system limit or option (both are referred to as variables).

The following table lists each configurable variable and its corresponding *name*, which represents the system variable to be queried. The names come from the files **<limits.h>** and **<unistd.h>**; the variables are defined in the referenced POSIX standards.

If successful, **sysconf()** returns the current value of the variable.

If *name* is not associated with a supported variable, **sysconf()** returns -1 . If *name* is not a valid value, **sysconf()** returns -1 and sets *errno*.

Table 45. Configurable System Variables

Variable	<i>name</i> Value
{ARG_MAX}	{_SC_ARG_MAX}
{CHILD_MAX}	{_SC_CHILD_MAX}
clock ticks per second	{_SC_CLK_TCK}
{NGROUPS_MAX}	{_SC_NGROUPS_MAX}
{OPEN_MAX}	{_SC_OPEN_MAX}
{STREAM_MAX}	{_SC_STREAM_MAX}
{TZNAME_MAX}	{_SC_TZNAME_MAX}
{_POSIX_JOB_CONTROL}	{_SC_JOB_CONTROL}
{_POSIX_SAVED_IDS}	{_SC_SAVED_IDS}
{_POSIX_VERSION}	{_SC_VERSION}
{_POSIX_AIO_LISTIO_MAX}	{_SC_AIO_LISTIO_MAX}
{_POSIX_AIO_MAX}	{_SC_AIO_MAX}
{_DELAYTIMER_MAX}	{_SC_DELAYTIMER_MAX}

sysconf()

Table 45. Configurable System Variables

Variable	<i>name</i> Value
{_MQ_OPEN_MAX}	{_SC_MQ_OPEN_MAX}
{_MQ_PRIO_MAX}	{_SC_MQ_PRIO_MAX}
{PAGESIZE}	{_SC_PAGESIZE}
{RTSIG_MAX}	{_SC_RTSIG_MAX}
{SEM_NSEMS_MAX}	{_SC_SEM_NSEMS_MAX}
{SEM_VALUE_MAX}	{_SC_SEM_VALUE_MAX}
{_TIMER_MAX}	{_SC_TIMER_MAX}
{_POSIX_ASYNCHRONOUS_IO}	{_SC_ASYNCHRONOUS_IO}
{_POSIX_MAPPED_FILES}	{_SC_MAPPED_FILES}
{_POSIX_MEMLOCK}	{_SC_MEMLOCK}
{_POSIX_MEMLOCK_RANGE}	{_SC_MEMLOCK_RANGE}
{_POSIX_MEMORY_PROTECTION}	{_SC_MEMORY_PROTECTION}
{_POSIX_MESSAGE_PASSING}	{_SC_MESSAGE_PASSING}
{_POSIX_PRIORITIZED_IO}	{_SC_PRIORITIZED_IO}
{_POSIX_PRIORITY_SCHEDULING}	{_SC_PRIORITY_SCHEDULING}
{_POSIX_REALTIME_SIGNALS}	{_SC_REALTIME_SIGNALS}
{_POSIX_SEMAPHORES}	{_SC_SEMAPHORES}
{_POSIX_FSYNC}	{_SC_FSYNC}
{_POSIX_SHARED_MEMORY_OBJECTS}	{_SC_SHARED_MEMORY_OBJECTS}
{_POSIX_SYNCHRONIZED_IO}	{_SC_SYNCHRONIZED_IO}
{_POSIX_TIMERS}	{_SC_TIMERS}
{_SIGQUEUE_MAX}	{_SC_SIGQUEUE_MAX}
{_POSIX_THREADS}	{_SC_THREADS}
{_POSIX_REENTRANT_FUNCTIONS}	{_SC_REENTRANT_FUNCTIONS}
{_POSIX_THREAD_ATTR_STACKSIZE}	{_SC_THREAD_ATTR_STACKSIZE}
{_POSIX_THREAD_PRIORITY_SCHEDULING}	{_SC_THREAD_PRIORITY_SCHEDULING}

Table 45. Configurable System Variables

Variable	<i>name</i> Value
{_POSIX_THREAD_PRIO_INHERIT}	{_SC_THREAD_PRIO_INHERIT}
{_POSIX_THREAD_PRIO_PROTECT}	{_SC_THREAD_PRIO_PROTECT}
{_POSIX_THREADS_PROCESS_SHARED}	{_SC_THREADS_PROCESS_SHARED}
{PTHREAD_STACK_MIN}	{_SC_THREAD_STACK_MIN}
{PTHREAD_THREADS_MAX}	{_SC_THREAD_THREADS_MAX}
{_POSIX_THREAD_ATTR_STACKADDR}	{_SC_THREAD_ATTR_STACKADDR}
{_POSIX_THREAD_ATFORK}	{_SC_THREAD_ATFORK}
Maximum size of ttyname_r() name buffer	{_SC_TTYNAME_R_SIZE_MAX}
{PTHREAD_DESTRUCTOR_ITERATIONS}	{_SC_THREAD_DESTRUCTOR_ITERATIONS}
Maximum size of getpwuid_r() name buffer	{_SC_GETPW_R_SIZE_MAX}
Maximum size of getlogin_r() name buffer	{_SC_GETLOGIN_R_SIZE_MAX}
Maximum size of getgrgid_r() and getgrnam_r() data buffers	{_SC_GETGR_R_SIZE_MAX}
{PTHREAD_KEYS_MAX}	{_SC_THREAD_KEYS_MAX}

Errors

[EINVAL] *name* is not valid.

Example

See **fs_init()**, p. 231.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §4.8.1
- IEEE Draft Standard 1003.4/D13, §4.8.1.2
- IEEE Draft Standard 1003.4a/D8, §4.8.1.2

sysconfig()

Synopsis

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

int sysconfig( unsigned long action, void (*svc(void *arg)), unsigned long
               *svc_number);
```

Library

rtx.o

Description

sysconfig() installs and removes supervisor service functions and controls access to supervisor service functions.

If *action* is set to **SVC_INSTALL**, the function to be installed as a supervisor service function is specified in *svc* and a supervisor service number assigned to the function is returned in a variable pointed to by *svc_number*. If *action* is set to **SVC_REMOVE**, then supervisor service function specified in the variable pointed to by *svc_number* is removed from use. Only supervisor service functions installed using **sysconfig()** can be removed.

If *action* is set to **SVC_PRMASK**, the supervisor service function specified in the variable pointed to by *svc_number* can be successfully executed only by a thread in the **PTHREAD_PRIVILEGED_NP** state. If the thread is not in the **PTHREAD_PRIVILEGED_NP** state, the **EPERM** error is returned when the function is called. If *action* is set to **SVC_PRUNMASK**, all threads can execute the supervisor service function identified by value in the variable pointed to by *svc_number*. The action **SVC_INSTALL_RESTORE_EE** preserves the state of the MSR[EE] bit across the system call interface, allowing applications to create supervisor functions that can be interrupted by external interrupts.

Supervisor service function code and data must be allocated in **VM_PROTECT** storage.

The thread executing this call must be in **PTHREAD_PRIVILEGED_NP** state.

Note: **sysconfig()** is a valid call for OS Open without virtual memory, but many options have no meaning in that environment.

If successful, **sysconfig()** returns 0. Otherwise, **sysconfig()** returns an error number.

Errors

[ENOSPC]	No space left to set up new system call.
[EPERM]	Operation not permitted.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	No

References

- **pthread_create()**, p. 552
- **pthread_tgcreate_np()**, p. 598

system()

Synopsis

```
#include <stdlib.h>
int system ( const char *string );
```

Library

None. This is a macro defined in **<stdlib.h>**.

Description

system() returns -1 .

system() is provided for ANSI C compatibility.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.4.5

Synopsis

```
#include <math.h>
double tan( double x );
```

Library

mathLib.a

Description

tan() returns the tangent of x (measured in radians).

If the result is out range, **tan()** sets *errno*.

Errors

[ERANGE] x is out of range.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.2.7

tanh()

Synopsis

```
#include <math.h>
double tanh( double x );
```

Library

mathLib.a

Description

tanh() returns the hyperbolic tangent of *x*.

If the result is out range, **tanh()** sets *errno*.

Errors

[ERANGE]	The result is out of range.
----------	-----------------------------

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.5.3.3

Synopsis

```
#include <termios.h>
int tcdrain( int fildes );
```

Library

ttyLib.a

Description

tcdrain() waits until all output written to the tty object named by *fildes* has been transmitted.

If successful, **tcdrain()** returns 0. Otherwise, **tcdrain()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>fildes</i> is not a valid file descriptor.
[EINTR]	A signal interrupted tcdrain() .
[ENOTTY]	The file associated with <i>fildes</i> is not a terminal.

Example

This code fragment shows draining output for the standard output.

```
#include <termios.h>
#include <unistd.h>
:
:
int rc;
:
:
rc = tcdrain(STDOUT_FILENO);
:
:
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, § 7.2.2
- **tcflow()**, p. 850
- **tcflush()**, p. 852
- **tcsendbreak()**, p. 858

tcflow()

Synopsis

```
#include <termios.h>
int tcflow( int fildes, int action );
```

Library

ttyLib.a

Description

tcflow() suspends data reception on the tty object named by *fildes*, depending on *action*:

TCOOFF	Suspending output is not supported.
TCOON	Restarting suspended output is not supported.
TCIOFF	OS Open transmits a STOP character to cause the terminal device to stop transmitting data.
TCION	OS Open transmits a START character to cause the terminal device to start transmitting data.

Symbolic constants for *action* are defined in **<termios.h>**.

If successful, **tcflow()** returns 0. Otherwise, **tcflow()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>fildes</i> is not a valid file descriptor.
[EINTR]	A signal interrupted tcflow() .
[ENOTTY]	The file associated with <i>fildes</i> is not a terminal.
[ENOTSUP]	The action is not supported by OS Open.

Example

This code fragment shows stopping and restarting the standard input.

```
#include <termios.h>
#include <unistd.h>
:
:
int rc;
:
:
rc = tcflow(STDIN_FILENO, TCIOFF);
:
:
rc = tcflow(STDIN_FILENO, TCION);
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, § 7.2.2
- **tcdrain()**, p. 849
- **tcflush()**, p. 852
- **tcsendbreak()**, p. 858

tcflush()

Synopsis

```
#include <termios.h>
int tcflush( int fildes, int queue_selector );
```

Library

ttyLib.a

Description

tcflush() discards any data written to the tty object named by *fildes* but not transmitted, or data received but not read, depending on *queue_selector*.

TCIFLUSH	It flushes data received but not read.
TCOFLUSH	It flushes data written but not transmitted.
TCIOFLUSH	It flushes both data received but not read and data written but not transmitted.

The symbolic constants for *queue_selector* are defined in **<termios.h>**.

If successful, **tcflush()** returns 0. Otherwise, **tcflush()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>fildes</i> is not a valid file descriptor.
[EINTR]	A signal interrupted tcflush() .
[ENOTTY]	The file associated with <i>fildes</i> is not a terminal.

Example

This code fragment shows flushing both input and output data for the terminal associated with the standard input.

```
#include <termios.h>
#include <unistd.h>
:
:
int rc;
:
:
rc = tcflush(STDIN_FILENO, TCIOFLUSH);
:
:
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, § 7.2.2
- **tcdrain()**, p. 849
- **tcflow()**, p. 850
- **tcsendbreak()**, p. 858

tcgetattr()

Synopsis

```
#include <termios.h>
int tcgetattr( int fildes, struct termios *termios_p );
```

Library

ttyLib.a

Description

tcgetattr() gets the parameters associated with the object named by *fildes* and stores them in the **termios** structure referenced by *termios_p*.

If successful, **tcgetattr()** returns 0. Otherwise, **tcgetattr()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>fildes</i> is not a valid file descriptor.
[ENOTTY]	The file associated with <i>fildes</i> is not a terminal.

Example

This code fragment shows turning echo off to enter a password.

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

/*
 * Ask for user's password
 */
char *getPasswd(char *str) {
    int rc;
    struct termios ttyAttr;

    rc = tcgetattr(STDIN_FILENO, &ttyAttr);
    if ( rc != 0 )
        return NULL;
    ttyAttr.c_lflag &= ~ECHO;
    rc = tcsetattr(STDIN_FILENO, TCSANOW, &ttyAttr);
    if ( rc != 0 )
        return NULL;
    printf("Please enter your password\n");
    gets(str);
    ttyAttr.c_lflag |= ECHO;
    tcsetattr(STDIN_FILENO, TCSANOW, &ttyAttr);
    return str;
} /* end getPasswd() */
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, § 7.2.1
- **tcsetattr()**, p. 860
- **tty_init()**, p. 915

tcgetkey()

Synopsis

```
#include <ttyLib.h>
char *tcgetkey( int fildes, int key, char *command );
```

Library

ttyLib.a

Description

tcgetkey() copies the string assigned to the function key, specified by *key*, to the buffer pointed to by *command* for the object named by *fildes*.

If successful, **tcgetkey()** returns *command*. Otherwise, **tcgetkey()** returns NULL and sets *errno*.

Errors

[EBADF]	<i>fildes</i> is not a valid file descriptor.
[ENOTTY]	The file associated with <i>fildes</i> is not a terminal.

Example

This function saves the command associated with function key “F1” and reassigns the key to the string “myHelp()”.

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

char F1_save[PFKSTR]

/*
 * Set function key F1 to myHelp()
 */
int setF1() {
    tcgetkey(STDIN_FILENO, 1, F1_save);
    tcsetkey(STDIN_FILENO, 1, "myHelp()");
} /* end setF1() */
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **tcsetkey()**, p. 862

Synopsis

```
#include <sys/tcpipLib.h>
#include <sys/utsname.h>

unsigned long tcpip_init( char *hostname, unsigned long num_clusters,
                        unsigned long num_mbufs );
```

Library

tcpipLib.a

Description

tcpip_init() initializes the TCP/IP library.

tcpip_init(), which must be called before any other routine from TCP/IP library, can be called only once during an instance of the OS Open system.

hostname specifies the name of the host. Maximum *hostname* length is specified as **_UNAME_LENGTH**, which is defined in **<sys/utsname.h>**.

num_clusters specifies a number, if clusters are to be allocated for use by the TCP/IP library. The TCP/IP library internally does not require that more than one cluster be allocated when the library is initialized. If clusters are not available, mbufs is used to send data. A device driver might require cluster availability for its own use. Each cluster contains 4096 bytes.

num_mbufs specifies a number, if memory buffers are to be allocated for use by the TCP/IP library. The number of memory buffers used by the TCP/IP library varies, depending on network utilization. Usually between 100 and 1000 memory buffers should be allocated. Each memory buffer is 128 bytes in length.

A minimum of one cluster and one memory buffer must be specified.

If successful, **tcpip_init()** returns 0. Otherwise, **tcpip_init()** returns -1.

Errors

None.

Example

See **enet_attach()**, p. 183.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	No

tcsendbreak()

Synopsis

```
#include <termios.h>
int tcsendbreak( int fildes, int duration );
```

Library

ttyLib.a

Description

tcsendbreak() transmits a continuous stream of zero-valued bits (a break) to the file associated with the file descriptor *fildes* for the number of milliseconds specified by *duration*.

If *duration* is 0, **tcsendbreak()** transmits zeros for at least 0.25 seconds.

If successful, **tcsendbreak()** returns 0. Otherwise, **tcsendbreak()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>fildes</i> is not a valid file descriptor.
[ENOTTY]	The file associated with <i>fildes</i> is not a terminal.

Example

This code fragment shows sending a break to the terminal associated with the standard output for 0.25 seconds.

```
#include <termios.h>
#include <unistd.h>
:
:
int rc;
:
:
rc = tcsendbreak(STDOUT_FILENO, 0)
:
:
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, § 7.2.2
- **tcdrain()**, p. 849
- **tcflow()**, p. 850
- **tcflush()**, p. 852

tcsetattr()

Synopsis

```
#include <termios.h>

int tcsetattr( int fildes, int optional_actions,
               const struct termios *termios_p );
```

Library

ttyLib.a

Description

tcsetattr() sets the parameters associated with the terminal (unless support is required from the underlying device that is not available) from the **termios** structure referenced by *termios_p*, depending on *optional_actions*.

TCSANOW	The change occurs immediately.
TCSADRAIN	The change occurs after all output written to <i>fildes</i> has been transmitted. This function should be used when changing parameters that affect the output.
TCSAFLUSH	The change occurs after all output written to <i>fildes</i> has been transmitted. All input that has been received, but not read, is discarded before the change is made.

The symbolic constants for *optional_actions* are defined in **<termios.h>**.

The zero baud rate, B0, terminates the connection. If B0 is specified as the output rate when **tcsetattr()** is called, the modem control lines are no longer asserted. Normally, this disconnects the line.

If the input baud rate is equal to the 0 in the **termios** structure when **tcsetattr()** is called, **tcsetattr()** changes the input rate to that specified by the output baud rate, as if the input rate had been set to the output rate by **cfsetispeed()**. This usage of 0 is obsolete.

tcsetattr() returns 0 if it could perform any of the requested actions, even if some could not be performed. It sets the attributes that are supported and leaves unchanged the attributes that are not supported. If no part of the request can be performed, **tcsetattr()** returns -1 and sets *errno* to [EINVAL]. A subsequent call to **tcgetattr()** returns the actual state of the terminal device.

Errors

[EBADF]	<i>filedes</i> is not a valid file descriptor.
[EINTR]	A signal interrupted tcsetattr() .
[EINVAL]	<i>optional_actions</i> is not a supported value, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.
[ENOTTY]	The file associated with <i>filedes</i> is not a terminal.

Example

See **tcgetattr()**, p. 854.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, § 7.2.1
- **tcgetattr()**, p. 854

tcsetkey()

Synopsis

```
#include <ttyLib.h>
int tcsetkey( int fildes, int key, char *command );
```

Library

ttyLib.a

Description

tcsetkey() associates a character string (containing up to 15 characters) with a function key (numbered 1 through 12) for the object named by *fildes*.

If successful, **tcsetkey()** returns 0. Otherwise, **tcsetkey()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>fildes</i> is not a valid file descriptor.
[ENOTTY]	The file associated with <i>fildes</i> is not a terminal.

Example

See **tcgetkey()**, p. 856.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **tcgetkey()**, p. 856

Synopsis

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

int telnetd_assignfunc(
    void      (*shell_fcnt_p)(void *arg),
    void      (*shell_restore_fcnt_p)(void *arg),
    int       (*shell_threadid_fcnt_p)(),
    int       (*shell_cancel_fcnt_p)() );
```

Library

telnetdLib.a

Description

telnetd_assignfunc() allows specification of a function other than **shell()** to be started when a user connects to an OS Open system. *shell_fcnt_p* points to a function that will be started as a thread when a user connects to an OS Open telnet server. If *shell_fcnt_p* is set to NULL, **shell()** will be started.

shell_restore_fcnt_p points to a function that will be started as a thread when the user disconnects from an OS Open telnet server. If this parameter is NULL, **shell()** will be started. This only occurs if a user function of **shell()** was running when the user connected to the system.

shell_threadid_fcnt_p points to a function that is called to determine *thread id* of the thread controlling the console at the time that the user connects to OS Open using the telnet server. The function pointed to by *shell_threadid_fcnt_p* should return *thread id*, or 0 if the thread controlling the console has exited or if no thread is controlling the console.

Any thread that is controlling the console needs to exit in response to a SIGTERM or SIGINT signal. If *shell_threadid_fcnt_p* is set to NULL, **shell_threadid()** will be called. The function pointed to by *shell_cancel_fcnt_p* returns *thread id* of the thread started by the OS Open telnet server when the user connected to the system. If the thread started by the OS Open telnet server is not running, the function pointed to by *shell_cancel_fcnt_p* should return 0. If *shell_cancel_fcnt_p* is set to NULL, **shell_threadid()** is called.

Note: The function started by the OS Open telnet server must exit in response to a SIGTERM and SIGINT signal. This function must ensure that these two signals are unmasked during function execution.

Errors

None.

telnetd_assignfunc()

Example

If **shell()** is already running on the controlling console, a spinner function will be started when a user connects to OS Open. When the connection is broken or the spinner exits, **shell()** will be restarted and resume control of the console.

```
(void)telnetd_assignfunc(spinner, NULL, spinner_threadid, NULL)
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **shell()**, p. 732
- **shell_threadid()**, p. 733
- **telnetd_assigntty()**, p. 865
- **telnetd_start()**, p. 866

Synopsis

```
#include <sys/tnetdLib.h>
int telnetd_assigntty(char *name);
```

Library

tnetdLib.a

Description

telnetd_assigntty() allows for changing the default tty device name used by the telnet daemon. If this function is not called, the telnet daemon will use the **/dev/tty0** device name when restoring the environment after a telnet session is over.

name should point to a valid, installed tty device.

Note: **telnetd_start()** must be called before this function can be executed.

If successful, **telnetd_assigntty()** returns 0. Otherwise, **telnetd_assigntty()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **telnetd_start()**, p. 866

telnetd_start()

Synopsis

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

```
int telnetd_start( unsigned long shell_stacksize, int tty_devhandle );
```

Library

tnetdLib.a

Description

telnetd_start() starts the telnet server daemon.

shell_stacksize is a value of the shell thread stack and should be at least 10KB.

tty_devhandle is the device handle for the tty device driver, which must be installed before **telnetd_start()** is called. If the tty device is open, it is assumed to provide file descriptors for **stdin**, **stdout**, and **stderr**. No device should use the names **/dev/pty0** and **/dev/tty_telnetd**, which the telnet daemon uses to install new devices.

telnetd_start() assumes that the tty device was installed using **/dev/tty0** as the name. If another name is used, **telnetd_assigntty()** needs to be called to specify the tty device name.

The **tcpiplib.a** and **netLib.a** libraries must be initialized before **telnetd_start()** is called.

If **shell()** is running, its priority must be lower than the priority of the telnet daemon, which is specified by **TELNET_DAEMON_PRIO** in the file **<sys/tnetdLib.h>**. The thread running **shell()** should terminate when **shell()** exits.

Note: Only one instance of the shell can run at a time. If a shell is running on the console and a user accesses the OS Open system using telnet, the shell running on the console terminates.

If successful, **telnetd_start()** returns 0. Otherwise, **telnetd_start()** returns -1.

Errors

None.

Example

The following example illustrates a main program starting the telnet daemon.

```
commd_t    commd={{(int(*)())open, {(int)"/dev/s0", O_RDWR,
                                asyncParityNone, async ParityOdd,
                                asyncStopBits1, asyncDataBits8, 9600}}};
```

```
struct termios defa ={BRKINT| IGNCR| ICRNL| IGNPAR| IXON, OPOST,
                     CLOCL | CREAD | CS8, ECHO | ECHOE | ECHOK |
                     ECHONL | ICANON | IEXTEN, {CTRL_D, CTRL_AT,
                     CTRL_H, CTRL_C, CTRL_U, CTRL_BSLASH, CTRL_Z,
                     CTRL_Q, CTRL_S, 0, 0, 0, 0, 0, 0}, B9600, 0};

int tty_h, devhandle, fd1, fd2, fd3;
pthread_attr_t shell_attr;
pthread_t shell_thread;

/*
Main
*/

void main()
{
/* Initialize I/O and install tick handler. ioLib_init() and timertick_install() are
platform-specific function calls. Check platform-specific function descriptions
for function descriptions.
*/

if (ioLib_init()) {
(void)s1dbprintf("ioLib_init() failed.\n");
(void)ppcHalt();
}

if (timertick_install()) {
(void)s1dbprintf("timertick_install() failed.\n");
(void)ppcHalt();
}

/*
Install device subsystem and async device driver
*/

if (dev_io_init(25, 25)) {
(void)s1dbprintf("dev_io_init() failed.\n");
(void)ppcHalt();
}

if (driver_install(&devhandle, async_init)) {
(void)s1dbprintf("driver_install() failed, async driver.\n");
(void)ppcHalt();
}

if (device_install("/dev/s0", CHRTYPE, devhandle, 1, 128, 128)) {
(void)s1dbprintf("device_install() failed, async device.\n");
(void)ppcHalt();
}

}
```

telnetd_start()

```
    if (driver_install(&tty_h, tty_init)) {
        (void)s1dbprintf("driver_install() failed, tty driver.\n");
        (void)ppcHalt();
    }

    if (device_install("/dev/tty0", DTYPE_TTY, tty_h, &commd, NULL,&defa)) {
        (void)s1dbprintf("device_install() failed, tty device.\n");
        (void)ppcHalt();
    }
}

/*
Open stdin, stdout, stderr
*/

fd1=open("/dev/tty0", O_RDONLY);
fd2=open("/dev/tty0", O_WRONLY);
fd3=open("/dev/tty0", O_WRONLY);
if (fs_init()) {
    (void)s1dbprintf("fs_init() failed.\n");
    (void)ppcHalt();
}

/*
Initialize TCP/IP. tok_attach() is a platform-specific function
*/

if (tcpip_init("tyrlik", 1, 500)) {
    printf("tcpip_init(), failed\n");
    (void)ppcHalt();
}

if (net_init()) {
    (void)printf("net_init(), failed\n");
    (void)ppcHalt();
}

if (tok_attach(0, 16, 3)) {
    (void)printf("tok_attach(), failed\n");
    (void)ppcHalt();
}

if (ifconfig("tok0 tyrlik netmask 255.255.240.0")) {
    (void)printf("ifconfig(), failed\n");
    (void)ppcHalt();
}

/*
Set up shell_attr
*/
```

```
(void)pthread_attr_init(&shell_attr);
(void)pthread_attr_setstacksize(&shell_attr, 10* 1024);
(void)pthread_attr_setfp_np(&shell_attr, PTHREAD_FP_AVAILABLE_NP);
/*
Start shell
*/
if (pthread_create(&shell_thread, &shell_attr, shell, NULL)) {
    (void)printf("pthread_create(), failed for shell thread\n");
}
/*
Start telnet daemon.
*/
if (telnetd_start(10* 1024, tty_h)) {
    (void)printf("telnet_start(), failed\n");
}
return;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **net_init()**, p. 468
- **shell()**, p. 732
- **tcPIP_init()**, p. 857
- **telnetd_assigntty()**, p. 865

name as *host:file* where *host* is the host name and *file* is the fully qualified remote file name

- **mode** *type* - *type* should be one of **ascii**, **netascii**, **binary**, **image**, or **octet**. **ascii** and **netascii** are equivalent and specify a 7 bit ascii transfer. **binary**, **image**, and **octet** are equivalent and specify a byte for byte transfer.
- **put** *Local file Remote file* - places the file *Local file* from the local host to the remote host as *Remote file*.
- **quit** - exit the **tftp** session
- **rexmt** *rval* - set per-packet retransmission timeout
- **status** - shows the current status of the **tftp** program
- **timeout** *tval* - sets the total transmission timeout to *tval* seconds.
- **trace** - toggles packet tracing
- **verbose** - toggles verbose mode, which supplies additional details during file transfer.

Command line form of tftp

At the OpenShell prompt or from an application program, supply the following routine invocation:

```
➔➔➔ tftp(" -g local file — host — rmt file [image|ascii] ") ➔➔➔
      |   |
      |   +-----p-----
```

- **-g** retrieves (gets) *rmt file* from *host* and places it into *local file*
- **-p** places (puts) local file to *host* in *rmt file*
- **image** specifies a binary transfer
- **ascii** specifies an ascii transfer (this is also the default)

If successful, **tftp()** returns 0. Otherwise, **tftp()** returns -1.

Errors

None.

Example

The following example illustrates file transfers to the “mama” system. Commands entered by the user are highlighted. “//” indicates commentary text.

```
// Interactive session with host “mama”
OS OPEN>tftp(“mama”)
```

tftp()

```
tftp> binary
tftp> status
Connected to mama.
Mode: octet Verbose: off Tracing: off
Rexmt-interval: 5 seconds, Max-timeout: 25 seconds
// Retrieve file /u/somedir/test.file from "mama" as htest.fil
tftp> get /u/somedir/test.file htest.fil
Received 752 bytes in 0.21 seconds
// Place htest.fil on local system as /u/somedir/rmt.fil on "mama"
tftp> put htest.fil /u/somedir/rmt.file
Sent 752 bytes in 0.26 seconds
tftp> quit
// Retrieve file /u/somedir/test.file from "mama" as htest.fil
// using the command line form. Note the default is ascii
OS OPEN>tftp("-g htest.fil mama /u/somedir/test.file")
Received 789 bytes in 0.25 seconds
OS OPEN>
```

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

References

- **tftp()**, p. 870

Synopsis

```
#include <arpa/tftp.h>
unsigned long tftp_init();
```

Library

tftp.o

Description

tftp_init() initializes tftp support, allowing **tftp()** to be called by an Application thread group.

tftp_init() must be called before any use of **tftp()** and can be called only once.

If successful, **tftp_init()** returns 0. Otherwise, **tftp_init()** returns -1.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	No

References

- **tftp_init()**, p. 873

tg_dump()

Synopsis

```
#include <kadtLib.h>
int tg_dump(tg_t tgid, char *buf);
```

Library

kadtLib.a

Description

tg_dump() provides information about the thread group identified by *tgid*. If *tgid* is 0, the current thread group is used.

If *buf* is NULL, the output is generated by **bprintf()**. Otherwise the information is stored in the buffer pointed to by *buf*.

Note: The buffer pointed to by *buf* should contain at least 4000 bytes.

tg_dump() returns -1 if *tgid* is not a valid thread group ID. Otherwise **tg_dump()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No
Callable from App. Thread	No

Example

The following example shows a sample dump of the current thread group.

```
OS OPEN> tg_dump(0,NULL)
```

```
-----
Thread Group: 0xdf614   (Kernel thread group)

  Thread   Starting function                State
  0x10e570 0x69674    shell                                READY
  0x10f6e4 0x9aac0    rslid                               BLOCKED ON COND
  0x1202b0 0x573dc    biosenet_rcv_thread                BLOCKED IN WAIT
  0xffff9104 0x3fdb4    arptimer                           BLOCKED IN WAIT
  0xffff9508 0x405d4    net_setsoftnet_thread              BLOCKED IN WAIT
  0x1052e0 0x86b4c    async_error_thread                 BLOCKED IN WAIT
  0xffff20e4 0x883c4    select_thread                       BLOCKED IN WAIT
  0x106c48 0x332a4    main                                BLOCKED ON JOIN
  Timer      Owner
  0xbc38     0x10e570  shell                                Running
                                         NO
```

Virtual Address	Real Address	Size	Attributes
0xfffff000	0xfffff000	0x1000	0x80000241
0xfffe9000	0x003f4000	0x15000	0x80000e11
0xfffe0000	0xfffe0000	0x1000	0x80000e41
0x40000000	0x40000000	0x1000	0x80000641
0x0010e000	0x003c6000	0x33000	0x80000e11
0x00105000	0x003ff000	0x2000	0x80000e11
0x00021000	0x00021000	0xe4000	0x80000e51
0x00018000	0x003cb000	0x4000	0x80000611
0x00017000	0x003cb000	0x1000	0x00000611
0x00003000	0x003df000	0x14000	0x80000e11
0x00002000	0x00002000	0x1000	0x80000e51
0x00000000	0x00000000	0x2000	0x80000a51
Semaphore : 0x120660			
Message Q : 0x10e0ac			
Message Q : 0x10e9bc			

References

- `tg_dump_all()`, p. 876
- `thread_list()`, p. 884

tg_dump_all()

Synopsis

```
#include < kadtLib.h>
int tg_dump_all( char *buf);
```

Library

kadtLib.a

Description

tg_dump_all() provides information about all thread groups.

If *buf* is NULL, the output is generated by **bprintf()**. Otherwise, the information is stored in the buffer pointed to by *buf*.

Note: The buffer pointed to by *buf* should contain at least 4000 bytes.

tg_dump() returns -1 if *tgid* is not a valid thread group id. Otherwise **tg_dump()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No
Callable from App. Thread	No

Example

The following example shows a sample dump of all thread groups.

```
OS OPEN> tg_dump_all(NULL)
-----
Thread Group: 0x11290
  Thread   Starting function      State
  0xbdb8  0x10006fc4  my_hello_world  SUSPENDED
Virtual Address  Real Address    Size  Attributes
0x10013000      0x00011290    0x1000  0x00000612
0x1000b000      0x00380000    0x8000  0x80000612
0x1000a000      0x00380000    0x1000  0x00000612
0x10005000      0x00384000    0x5000  0x80000e12
Thread Group: 0xdf614  (Kernel thread group)
  Thread   State      Starting function
0x10e570  READY      0x69674  shell
0x10f6e4  BLOCKED ON COND 0x9aac0  rslid
0x1202b0  BLOCKED IN WAIT 0x573dc  biosenet_rcv_thread
```

```
0xfffe9104 BLOCKED IN WAIT 0x3fdb4  arptimer
0xfffe9508 BLOCKED IN WAIT 0x405d4  net_setsoftnet_thread
    0x1052e0 BLOCKED IN WAIT 0x86b4c  async_error_thread
0xfffe20e4 BLOCKED IN WAIT 0x883c4  select_thread
    0x106c48 BLOCKED ON JOIN 0x332a4  main

Timer                Owner
0xbc38              0x10e570  shell

Virtual Address  Real Address      Size  Attributes
0xfffff000      0xfffff000      0x1000  0x80000241
0xfffe0000      0xfffe0000      0x1000  0x80000e41
0x40000000      0x40000000      0x1000  0x80000641
0x0010e000      0x003c6000     0x33000  0x80000e11
0x00021000      0x00021000     0xe4000  0x80000e51
0x00003000      0x003df000     0x14000  0x80000e11
0x00002000      0x00002000     0x1000  0x80000e51
0x00000000      0x00000000     0x2000  0x80000a51

Semaphore : 0x120660
Message Q  : 0x10e0ac
```

References

- tg_dump(), p. 874

tg_list()

Synopsis

```
#include < kadtLib.h>
int tg_list( char **bufp );
```

Library

kadtLib.a

Description

tg_list() lists all thread groups and identifies the kernel thread group.

If *bufp* is NULL, the output is generated by **bprintf()**. Otherwise, the information is stored in the buffer pointed to by **bufp*.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No
Callable from App. Thread	No

Example

The following example shows a sample of **tg_list()** output.

```
OS OPEN> tg_list(NULL)
Thread Group
0x11290
0xdf614 (Kernel thread group)
----- Total Number of Thread Groups: 2 -----
```

Synopsis

```
#include <kadtLib.h>

int thread_attr_dump( pthread_attr_t * attrid, char *buf );
```

Library

kadtLib.a

Description

thread_attr_dump() provides information about the thread attributes object pointed to by *attrid*.

If *buf* is NULL, the output is generated by **bprintf()**. Otherwise, the information is stored in the buffer pointed to by *buf*.

thread_attr_dump() returns -1 if *attrid* does not point to a valid thread attribute. Otherwise, **thread_attr_dump()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

thread_debug()

Synopsis

```
#include <dbLib.h>

int thread_debug( pthread_t *thread, pthread_attr_t *attr,
                 unsigned long funcaddr, void *arg );
```

Library

dbLib.a

Description

thread_debug() creates a thread with a breakpoint set at the beginning of the supplied function.

The field pointed to by *thread* is set to the thread ID, unless *thread* is NULL. The thread is created with the attributes specified by the attribute object pointed to by *attr*. If *attr* is NULL, default attributes are used. The thread starts executing at the address specified by *funcaddr* and is passed *arg* as a parameter.

A breakpoint is set at *funcaddr* and the special shell variable **\$dfthread** is set to the thread ID of the newly created thread.

thread_debug() returns the return code from the calling **pthread_create()**.

Note: **thread_debug()** is intended for use from an OpenShell command line. If called in an application program, the behavior of **thread_debug()** is undefined.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

Synopsis

```
#include <kadtLib.h>
int thread_dump( pthread_t thid, char *buf );
```

Library

kadtLib.a

Description

thread_dump() provides information about the thread identified by *thid*.

If *buf* is NULL, the output is generated by **bprintf()**. Otherwise, the information is stored in the buffer pointed to by *buf*.

Note: This buffer should have a minimum capacity of 1200 bytes.

thread_dump() returns -1 if *thid* is not a valid thread ID. Otherwise, **thread_dump()** returns 0.

Errors

None.

Example

The following shows a sample dump of thread 0x425820.

```
OS OPEN>thread_dump(0x425820)
```

```
-----
```

Thread ID: 0x425820

Starting function: 0x26dbfc (shell)

Thread state: READY errno: (0) Operation successful(ESUCCESS)

Stacksize: 40960 Detach state: JOINABLE Contention scope: PROCESS

Inherit sched: DEFAULT Scheduler: FIFO Priority: 28 Base priority: 28

Registers:

00 00000001 00424f80 00273e38 00000000 80000000 00000000 00000000
00008000

08 00004000 0000c000 00000013 00172100 88008028 0027f8dc 00000000
0027f8cc

16 0027f668 0027f8c8 0027f668 0027f410 0027f8d0 0016b890 0027f8d4
0016eea0

24 ffff7fff 004bdbe8 0000004c 004bdbe0 004ba9ff 004bdbd8 004baa00
0000004d

xer: 0000000e lr: 00030834 ctr: 00000013

thread_dump()

cr: 28008022 srr0: 0005fa88 srr1: 00009000

Cancel state: NONE Interrupt state: ENABLE Interrupt type: CONTROLLED

Floating-Point: ENABLED

Signals pending: 0x00000000 Signal mask: 0x00000422

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84

Synopsis

```
#include <kadtLib.h>
int thread_info_list( char **bufp );
```

Library

kadtLib.a

Description

thread_info_list() provides formatted information about active threads.

The information is part of the **kda_dump()** display.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

thread_info_list() returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84
- **kda_dump()**, p. 349

thread_list()

Synopsis

```
#include <kadtLib.h>
int thread_list( char *buf );
```

Library

kadtLib.a

Description

thread_list() lists all threads active in OS Open.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

thread_list() returns 0.

Errors

None.

Example

The following shows a sample listing of active threads.

```
OS OPEN>thread_list()
```

Thread	Starting Function	State
0x425820	0x26dbfc shell	READY
0x426c98	0x26dc2c telnetd	BLOCKED ON COND
0x42bb7c	0x26d628 ftpd	BLOCKED ON COND
0x468c58	0x26d334 rsld	BLOCKED ON COND
0x46d910	0x26ce24 enet_rcv_thread	BLOCKED ON COND
0x471d98	0x26cc50 slow_timeout_thread	BLOCKED IN WAIT
0x475f44	0x26cc44 fast_timeout_thread	BLOCKED ON WAIT
0x47a0f0	0x26cde8 slow_timeout_thread_if	BLOCKED IN WAIT
0x4aa7f4	0x26ce00 arptimer	BLOCKED IN WAIT
0x4ae9a0	0x26cdf4 net_setsoftnet_thread	BLOCKED IN WAIT
0x4b9474	0x26d0b0 motor_control	BLOCKED IN WAIT
0x4bed70	0x26cc08 select_thread	BLOCKED IN WAIT

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

Synopsis

```
#include <kadtLib.h>

char *thread_name( pthread_t thid, char *name );
```

Library

kadtLib.a

Description

thread_name() copies the name of the starting function of a thread, identified by *thid*, into the buffer pointed to by *name*.

If successful, **thread_name()** returns *name*. If *thid* is not a valid thread, **thread_name()** copies the string “invalid” to *name*. If the thread is valid, but the name cannot be found, the string “???” is copied to *name*.

Note: The buffer pointed to by *name* should have a capacity of at least 40 characters.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

time()

Synopsis

```
#include <time.h>
time_t time( time_t *timeptr );
```

Library

cLib.a

Description

time() returns the current calendar time as the number of seconds since January 1, 1970 00:00:00 universal coordinated time (UTC).

The return value is also stored in the location pointed to by *timeptr*. If *timeptr* is NULL, the return value is not stored.

The value (*time_t*)(-1) is returned if the calendar time is not available.

Note: The real time clock must be set at least once using **clock_settime()**. Otherwise, the results of **time()** are undefined.

Errors

None.

Example

See **asctime()**, p. 62.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.12.2.4
- **asctime()**, p. 62
- **clock_settime()**, p. 126
- **ctime()**, p. 149
- **gmtime()**, p. 299
- **localtime()**, p. 388
- **mktime()**, p. 429

Synopsis

```
#include <netLib.h>
#include <time.h>

int time_request(char *hostname, char *protocol, time_t *server_time);
```

Library

netLib.a

Description

time_request() retrieves current time from server specified by *hostname* parameter. The protocol used for this transaction is specified in protocol parameter. Protocol can be either set to "tcp" or "udp".

If *server_time* is NULL, then OS Open **CLOCK_REALTIME** is set to the value returned from the server. Otherwise location pointed to by *server_time* is updated with the current time value returned from the server.

When successful, **time_request()** returns 0, otherwise -1 is returned.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

timer_create()

Synopsis

```
#include <signal.h>
#include <time.h>

int timer_create( clockid_t clock_id, struct sigevent *evp,
                  timer_t *timerid );
```

Library

rtxLib.a

Description

timer_create() creates a timer using the specified clock, *clock_id*, as the timing base.

In the OS Open system, *clock_id* must be **CLOCK_REALTIME**.

A thread can create one or more timers. Each timer is associated with the thread that creates it. When a timer expires, a signal is sent to the associated thread.

evp points to a **sigevent** structure that specifies the signal number to be sent each time the timer expires. If *evp* is NULL, the default signal, **SIGALARM**, is sent.

A timer is disarmed and inactive when it is created. **timer_settime()** can arm and disarm a timer.

If successful, **timer_create()** returns 0 and stores the identifier of the newly created timer in the field pointed to by *timerid*. Otherwise, **timer_create()** returns -1 and sets *errno*.

Errors

[ENOMEM]	Insufficient memory exists to create the timer.
[EINVAL]	The specified clock is not defined.

Example

This example reads the characters of an escape sequence from the standard input. A timer ensures that all characters have been read in the escape sequence.

```
#include <stdio.h>
#include <signal.h>
#include <stddef.h>
#include <time.h>

char *getESCchars(char *escape_buffer)
{
    char *wp;
    char ch;
    timer_t timerid;
```



```
int sig=0;
itimerspec_t v, o;
sigset_t set;

/* Initialize timer structure for 10 milliseconds */
v.it_interval.tv_sec = 0;
v.it_interval.tv_nsec = 0;
v.it_value.tv_sec = 0;
v.it_value.tv_nsec = 100000000; /* 10 milliseconds */
/* create timer and start it */
timer_create(CLOCK_REALTIME, NULL, &timerid);
timer_settime(timerid, TIMER_RELTIME_NP, &v, &o);
/* read in escape sequence characters */
wp = escape_buffer;
do
{
    while(*wp != '\0')
        ch = getchar();
    if(ch > 0)
        *wp++ = ch;
    else
        *wp = '\0';
}
/* receive the SIGALRM signal and delete timer */
sigfillset(&set);
sigwait(&set, sig);
timer_delete(timerid);
return(escape_buffer);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Draft Standard 1003.4/D13, §14.2.2
- **sigwait()**, p. 751
- **timer_delete()**, p. 890
- **timer_settime()**, p. 897

timer_delete()

Synopsis

```
#include <signal.h>
#include <time.h>

int timer_delete( timer_t timerid );
```

Library

rtxLib.a

Description

timer_delete() deletes the timer specified by *timerid*.

If the timer is armed when **timer_delete()** is called, the timer is deleted, the timer does not expire, and no signal is generated.

timer_delete() does not affect signals that are pending for the associated thread. Pending signals remain pending.

If successful, **timer_delete()** returns 0. Otherwise, **timer_delete()** returns -1 and sets *errno*.

Errors

[EINVAL]	The timer specified by <i>timerid</i> does not refer to a currently existing timer.
----------	---

Example

See **timer_create()**, p. 888.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §14.2.3
- **timer_create()**, p. 888

Synopsis

```
#include <kadtLib.h>

int timer_dump( timer_t timerid, char *buf );
```

Library

kadtLib.a

Description

timer_dump() provides information about the timer block *timerid*.

If *buf* is NULL, the output is generated by **bprintf()**. Otherwise, the information is stored in the buffer pointed to by *buf*.

timer_dump() returns -1 if *timerid* is not a valid timer ID. Otherwise, **timer_dump()** returns 0.

Errors

None.

Example

The following example shows a sample dump for timer 0x41ae38.

```
OS OPEN>timer_dump(0x41ae38)
```

```
-----
Timer ID: 0x41ae38
Owner: 0x425820 (shell)
Not Running
-----
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84

timer_getoverrun()

Synopsis

```
#include <time.h>

int timer_getoverrun( timer_t timerid );
```

Library

rtxLib.a

Description

timer_getoverrun() returns the timer overrun count for the timer named by *timerid*.

Each time a timer expires, OS Open attempts to deliver the specified signal to the associated thread. If this signal is already pending for the associated thread, OS Open increments the timer's overrun count. The timer overrun count is set to zero when the timer is created and is reset to zero by **timer_getoverrun()**.

OS Open imposes a limit of **DELAYTIMER_MAX** on timer overrun counts. If a timer overrun count reaches this value, it is not incremented further. The timer overrun count keeps this value until it is reset by a call to **timer_getoverrun()**.

If successful, **timer_getoverrun()** returns the timer overrun count for the specified timer. Otherwise, **timer_getoverrun()** returns -1 and sets *errno*.

Errors

[EINVAL]	The timer specified by <i>timerid</i> does not refer to an existing timer.
----------	--

Example

See **timer_gettime()**, p. 893.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §14.2.4
- **sigwait()**, p. 751
- **timer_gettime()**, p. 893
- **timer_settime()**, p. 897

Synopsis

```
#include <time.h>

int timer_gettime( timer_t timerid,
                  itimerspec_t *value );
```

Library

rtxLib.a

Description

timer_gettime() returns the time until expiration and the reload value for the timer specified by *timerid*.

value points to a **itimerspec_t** structure. The fields in the **itimerspec_t** structure are updated as described below:

it_value If the timer is armed, this field is set to the amount of time before the timer expires. If the timer is unarmed, this field is set to zero. This value is returned as the relative time until expiration, even if the timer was armed with an absolute expiration time.

it_interval This field is set to the timer's reload value.

If successful, **timer_gettime()** returns 0. Otherwise, **timer_gettime()** returns -1 and sets *errno*.

Errors

[EINVAL] The timer specified by *timerid* does not refer to an existing timer.

Example

This example checks to see if a timer is armed or not. If the timer is disarmed, a zero is returned. Otherwise, a positive value is returned, and the timer's overrun count is cleared.

```
#include <time.h>

int istimer_running(timer_t timerid)
{
    itimerspec_t t;
    int rv = 1;
    /* Get current values for the timer */
    timer_gettime(timerid, &t);
    if((t.it_value.tv_sec == 0) && (t.it_value.tv_nsec == 0))
```

timer_gettime()

```
        rv = 0;
        /* Clear timer's overrun count */
        timer_getoverrun(timerid);
        return(rv);
    }
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §14.2.4
- **timer_getoverrun()**, p. 892
- **timer_settime()**, p. 897

Synopsis

```
#include <kadtLib.h>
int timer_list( char **bufp );
```

Library

kadtLib.a

Description

timer_list() provides formatted information about all timers.

The information is part of the **kda_dump()** display.

The information is printed using **bprintf()**. *bufp* is passed as the first parameter to **bprintf()**.

timer_list() returns 0.

Errors

None.

Example

The following example shows a sample timer listing.

```
OS OPEN>timer_list()

Timer      Owner      Running
0x41ae38   0x425820   shell      NO
0x425954   0x425820   shell      NO
0x426dcc   0x426c98   telnetd    NO
0x42bcb0   0x42bb7c   ftpd       NO
0x468d8c   0x46d910   rsld       YES
0x46da44   0x46d910   enet_rcv_thread  NO
0x471ecc   0x471d98   slow_timeout_thread  YES
0x476078   0x475f44   fast_timeout_thread  YES
0x47a224   0x47a0f0   slow_timeout_thread_if YES
0x4aa928   0x4aa7f4   arptimer   YES
0x4aead4   0x4ae9a0   net_setsoftnet_thread NO
0x4fff88   0x4fe54    thread0    NO

----- Total Number of Timers: 15 -----
```

timer_list()

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84
- **kda_dump()**, p. 349

Synopsis

```
#include <time.h>

int timer_settime( timer_t timerid,
                  int flags,
                  itimerspec_t *value,
                  itimerspec_t *ovalue );
```

Library

rtxLib.a

Description

timer_settime() arms or disarms the timer named by *timerid*.

The mutually exclusive values for *flags* are the following.

TIMER_ABSTIME Indicates absolute time

TIMER_RELTIME_NP Indicates relative time

value points to an **itimerspec_t** structure containing two fields. These fields are described below:

it_value Specifies the time of the next timer expiration. This time can be specified as an absolute time or a relative time. If the flag **TIMER_ABSTIME** is set in *flags*, then *it_value* is interpreted as an absolute time. Otherwise, *it_value* is interpreted as a relative time.

If the timer was armed when **timer_settime()** is called, the timer is effectively disarmed and then armed again with the new *it_value*.

If *it_value* specifies an absolute time and that time has passed, the timer expires immediately.

If the value of *it_value* is zero, the timer is disarmed. In this case, signals pending for the associated thread are not affected. Pending signals remain pending.

it_interval Specifies a reload value for the timer. OS Open remembers the reload value for each timer. When a timer expires, OS Open restarts the timer if its reload value is nonzero.

The reload value is used to create a periodic (or repetitive) timer. Note that the periodic intervals begin when the initial interval, specified by *it_value*, expires.

If *ovalue* is not NULL, it points to an **itimerspec_t** structure. The fields in the **itimerspec_t** structure are updated as described below:

timer_settime()

`it_value` If the timer was armed, this field is set to the amount of time before the timer would have expired. If the timer was unarmed, this field is set to zero. This value is returned as the relative time until expiration even if the timer was armed with an absolute expiration time.

`it_interval` This field is set to the previous value of the timer's reload interval.

If successful, **timer_settime()** returns 0. Otherwise, **timer_settime()** returns `-1` and sets *errno*.

Errors

[EINVAL] The timer specified by *timerid* does not refer to a currently existing timer. A **value** structure specified a nanosecond value less than zero or greater than or equal to 1000 million.

Example

See **timer_create()**, p. 888.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Draft Standard 1003.4/D13, §14.2.4
- **sigwait()**, p. 751
- **timer_getoverrun()**, p. 892
- **timer_gettime()**, p. 893

Synopsis

```
#include <flih.h>
void timertick_notify( void );
```

Library

rtxLib.a

Description

timertick_notify() notifies OS Open that a timer tick (as defined in the configuration block) expired.

OS Open uses **timertick_notify()** to maintain its timer reference for timer and time of day services.

OS Open systems must make arrangements to call **timertick_notify()** at timer-tick intervals for proper operation of OS Open timer-related services.

timertick_notify() returns nothing.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

tmpfile()

Synopsis

```
#include <stdio.h>
FILE *tmpfile ( void );
```

Library

fsLib.a

Description

tmpfile() creates a temporary binary file in the working directory that is removed when the file is closed.

The calling program must have write permission for the working directory.

The temporary file may not be removed if OS Open terminates without closing the file.

The file is opened for update in **wb+** mode.

If successful, **tmpfile()** returns a pointer to the stream of the created file. If the file cannot be created, **tmpfile()** returns a NULL pointer.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.4.3
- **fopen()**, p. 210
- **pthread_exit()**, p. 557

Synopsis

```
#include <stdio.h>
char *tmpnam( char *s );
```

Library

fsLib.a

Description

tmpnam() creates a unique valid file name.

If *s* is set to NULL, **tmpnam()** stores the name in an internal buffer that may be overwritten by subsequent calls to **tmpnam()**. If *s* is not null, **tmpnam()** writes the name in the location pointed to by *s* and returns the location.

Because names returned by **tmpnam()** are mechanical constructs that only accidentally have meaning, the names are usually used for temporary files.

tmpnam() generates up to **TMP_MAX** unique names per system initialization. If **tmpnam()** is called more than **TMP_MAX** times, returned file names are not guaranteed to be unique.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.4.4

tn()

Synopsis

```
#include <shell.h>
int tn( char *host );
```

Library

telnet.o

Description

tn() connects the local host with a remote host using the TELNET interface.

tn(), which uses the standalone I/O streams, is for use from an OpenShell command line.

If successful, **tn()** returns 0. Otherwise, **tn()** returns -1.

Errors

None.

Example

The following example illustrates connecting to the "pep0" system. Commands entered by the user are highlighted.

```
OS Open>tn("pep0")
```

```
Trying 9.67.192.9...
```

```
Connected to pep0.
```

```
Operating in single character mode
```

```
Local character echo
```

```
Escape character is '^['.
```

```
AIX telnet (pep0.raleigh.ibm.com)
```

```
IBM AIX Version 3 for RISC System/6000
```

```
(C) Copyrights by IBM and by others 1982, 1991.
```

```
login: ron
```

```
ron's Password:
```

```
*****
```

```
* *
```

```
* *
```

```
* Welcome to IBM AIX Version 3.2 !*
```

```
* *
```

```
* *
```

```
* Please see the README file in /usr/lpp/bos for information *
```

```
* pertinent to this release of the AIX Operating System.*
```

* *
* *

Last unsuccessful login: Tue Apr 13 09:24:25 EDT 1993 on pts/1 from kaboom.raleim

Last login: Wed Apr 28 08:44:13 EDT 1993 on pts/5 from osopen01.raleigh.ibm.com

pep0.raleigh.ibm.com[/u/ron]\$ **who**

ron pts/0 Apr 28 16:03 (osopen01.raleigh)
pep0.raleigh.ibm.com[/u/ron]\$ **lssrc -t telnet**

Service	Command	Description	Status
telnet	telnetd		active

Pid	Inet Address	Hostname
5978	9.67.194.3	osopen01.raleigh.ibm.com

terminal type = TERM=vt100
pep0.raleigh.ibm.com[/u/ron]\$ ^] tn> ?

Commands may be abbreviated. Commands are:

close	close current connection
logout	forcibly logout remote user and close the connection
display	display operating parameters
mode	try to enter line or character mode ('mode ?' for more)
open	connect to a site
quit	exit telnet
send	transmit special characters ('send ?' for more)
set	set operating parameters ('set ?' for more)
unset	unset operating parameters ('unset ?' for more)
status	print status information
toggle	toggle operating parameters ('toggle ?' for more)
slc	change state of special characters ('slc ?' for more)
environ	change environment variables ('environ ?' for more)
?	print help information

tn> **quit**

Connection closed.

OS Open>

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No

tok_arpinut()

Synopsis

```
#include <netinet/if_token.h>
#include <sys/mbuf.h>

void tok_arpinut( struct arpcom *ac, struct mbuf *m );
```

Library

tcpiLib.a

Description

tok_arpinut() is a token-ring address resolution protocol (ARP) input routine.

ac is a pointer to the **arpcom** structure that is shared between the token-ring device driver and ARP routines. *m* points to a buffer holding a raw token-ring ARP packet. A token-ring device driver should call **tok_arpinut()** when it receives an ARP packet.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **tok_arpresolve()**, p. 905

Synopsis

```
#include <netinet/if_token.h>
#include <sys/mbuf.h>

int tok_arpresolve( struct arpcom *ac, struct mbuf *m,
                    struct in_addr *destip, struct sockaddr_tr *dsttr );
```

Library

tcpipLib.a

Description

tok_arpresolve() resolves an IP address into a token-ring address.

If the resolution succeeds, *dsttr* is filled in with a token-ring destination address. *destip* contains a filled-in structure containing the destination IP address, and *m* contains the packet to be transmitted.

If the destination address is the same as the local address, the loopback interface is used for transmission if it is available.

If the address cannot be immediately resolved, an address resolution protocol (ARP) packet is transmitted, and the packet contained in *m* is queued for transmission when the address is resolved. This routine is called from the token-ring device driver output routine.

tok_arpresolve() returns 0 if the address cannot be immediately resolved.

tok_arpresolve() returns 1 if the address is resolved and the packet can be transmitted immediately.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **tok_arpinut()**, p. 904

tolower()

Synopsis

```
#include <ctype.h>
int tolower( int c );
```

Library

cLib.a

Description

tolower() converts *c* to lowercase if *c* represents an uppercase letter. Otherwise, *c* is unchanged.

Errors

None.

Example

See **getchar_unlocked()**, p. 258.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.2.1
- **islower()**, p. 340
- **toupper()**, p. 907

Synopsis

```
#include <ctype.h>
int toupper( int c );
```

Library

cLib.a

Description

toupper() converts *c* to uppercase if *c* represents a lowercase letter. Otherwise, *c* is unchanged.

Errors

None.

Example

This example modifies characters between code 0x20 and code 0x7e to uppercase.

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

void char_modf(void)
{
    int ch;
    for(ch = 0x20; ch <= 0x7e; ch++)
        printf("upper case of %c is %c\n", ch, toupper(ch));
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.3.2.2
- **isupper()**, p. 347
- **tolower()**, p. 906

trace_checkall()

Synopsis

```
#include <trace.h>
#include <dbLib.h>

int trace_checkall(unsigned long svc_number,
    unsigned long *number_calls, unsigned long *min_rtcl,
    unsigned long *max_rtcl, tb_t *total);
```

Library

dbLib.a

Description

trace_checkall() scans the entire OS Open kernel trace buffer for entries associated with the system call specified by *svc_number*. If the trace buffer contains entries for *svc_number*, then **trace_checkall()** assigns the number of found entries to the variable pointed to by *number_calls*.

Additionally, **trace_checkall()** stores minimum, maximum, and total execution times of a system call in variables pointed to by *min_rtcl*, *max_rtcl* and *total* respectively. All execution times are represented in units used by time base or real time clock registers of the processor that executes this call.

Total execution time can be used to compute average execution time for given system call.

If successful, **trace_checkall()** returns 0. Otherwise, it returns -1. If kernel trace is inactive or no trace entries for *svc_number* can be found, **trace_checkall()** fails.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **trace_checklast()**, p. 909
- **trace_copy()**, p. 910
- **trace_snapshot()**, p. 912

Synopsis

```
#include <trace.h>
#include <dbLib.h>

int trace_checklast(unsigned long svc_number,
                    unsigned long *time_rtcl);
```

Library

dbLib.h

Description

trace_checklast() scans the OS Open kernel trace buffer for an entry associated with the system call specified by *svc_number*. If the trace buffer contains an entry for *svc_number*, then **trace_checklast()** assigns time of execution of *svc_number* to a variable pointed to by *time_rtcl*. Only the most recent entry for *svc_number* is examined.

If successful, **trace_checklast()** returns 0. Otherwise, it returns -1. If kernel trace is not active or if no trace entries for *svc_number* can be found, **trace_checklast()** fails.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **trace_checkall()**, p. 908
- **trace_copy()**, p. 910
- **trace_snapshot()**, p. 912

trace_copy()

Synopsis

```
#include <dbLib.h>

int trace_copy(char *file)
```

Library

dbLib.h

Description

trace_copy() writes the OS Open kernel trace buffer into the file specified by *file*. **trace_copy()** allocates space for the trace buffer, copies the OS Open kernel trace buffer into this buffer, then performs the write. This assures that the trace buffer written to *file* contains all the calls issued prior to calling **trace_copy()**.

If successful, **trace_copy()** returns 0. Otherwise, **trace_copy()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **trace_checkall()**, p. 908
- **trace_checklast()**, p. 909
- **trace_snapshot()**, p. 912

Synopsis

```
#include <trace.h>

int trace_get( void **startaddr, unsigned long *size,
               void **current_location );
```

Library

rtxLib.a

Description

trace_get() retrieves trace information from the OS Open trace buffer.

The starting memory address of OS Open's trace buffer is stored in the address pointed to by *startaddr*. The current location of the latest trace entry is stored in the address pointed to by *current_location*. The size of the trace buffer is placed at the memory address pointed to by *size*. The size is the same as the size entered in the configuration block. If 0 is specified in the system configuration block, this function fails.

If successful, **trace_get()** returns 0. Otherwise, **trace_get()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

trace_snapshot()

Synopsis

```
#include <dbLib.h>
int trace_snapshot( char *buf );
```

Library

dbLib.a

Description

trace_snapshot() provides a snapshot of the OS Open trace buffer.

Note: The snapshot is a formatted representation of the data in the trace buffer. The formatted buffer information can be up to five times larger than the trace buffer.

If *buf* is NULL, the output is generated by **bprintf()**. If *buf* is not NULL, it must point to a buffer large enough to hold the output. It should be roughly five times larger than the trace size specified in the configuration block.

trace_snapshot() returns -1 if the trace is not active (the trace buffer size is 0). Otherwise, **trace_snapshot()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **bprintf()**, p. 84

Synopsis

```
#include <trace.h>

int trace_write( unsigned short id, unsigned long data1,
                unsigned long data2, unsigned long data3,
                unsigned long data4, unsigned long data5 );
```

Library

rtxLib.a

Description

trace_write() writes trace information into the OS Open trace buffer.

This function enters a **TRCHKG_ENTRY** trace entry into the OS Open trace buffer. A **TRCHKG_ENTRY** trace entry has the following format.

```
MMMTDDDD1111111122222222333333334444444455555555
```

All trace entry types are defined in the header file **<trace.h>**.

MMM and *T* in the trace format are constants written by **trace_write()**; *MMM* has the value X'D00' and *T* has the value X'6'.

DDDD, specified by the *id* parameter, is a hexadecimal number that represent a system or user-defined function call. The numbers X'0' through X'200' are reserved for the OS Open system. Some system functions are defined in the file **<trace.h>**. Numbers ranging from X'201' through X'FFFF' are available for user defined functions.

data1, *data2*, *data3*, *data4*, and *data5* are placed in the fields *11111111*, *22222222*, *33333333*, *44444444*, and *55555555*, respectively. These fields may contain any data that the user may need to know when a function is being traced, such as memory addresses, register values, variable values, and so on.

Note: The upper 12 bits of *data3* cannot be equal to any of the following values:
TRACE_KERNEL_CALL, TRACE_KERNELEXIT_CALL,
TRACE_EXTERNAL_INTERRUPT, TRACE_PROCESSOR_FAULT,
TRACE_DUMMY_ENTRY.

The above values are defined in **trace.h**.

If 0 was specified in the system configuration block for trace buffer size, this function fails.

If successful, **trace_write()** returns 0. Otherwise, **trace_write()** returns -1.

Errors

None.

Attributes

Async Safe	Yes
------------	-----

trace_write()

Cancel Safe	Yes
Interrupt Handler Safe	Yes

Synopsis

```
#include <ttyLib.h>

int tty_init( driver_t *dsw, va_list vargs );
```

Library

ttyLib.a

Description

tty_init() initializes the general terminal interface.

tty_init() should not be called directly but should be used as a parameter of **driver_install()**.

Errors

None.

Example

This code fragment shows the initial thread setting up a terminal as the standard input, the standard output, and the standard error. The terminal is attached to OS Open through async port 1 by using the asynchronous device driver.

```
#include <sys/devLib.h>
#include <sys/devDrive.h>
#include <sys/ioctl.h>
#include <sys.types.h>
#include <fcntl.h>
#include <ttyLib.h>
#include <termios.h>
#include <config.h>
#include <errno.h>
#include <sys/asyncLib.h>

void main();
void panic();
const conf_t _Kernel_Config_block = {
    main,                /* Main function */
    panic,               /* Panic function */
    0x00000000,          /* Main argc */
    0x00040000,          /* Main stack size */
    (void *)0x00000000,  /* Memory heap 1 start address */
    0x00500000,          /* Memory heap 1 size */
    0x00000000,          /* Memory heap 2 start address */
    0x00000000,          /* Memory heap 2 size */
    0x000004b0,          /* Trace table size */
    0x00fe5100,          /* Tick rate */
}
```

tty_init()

```
    0x0fe51000        /* Round robin time slice */
};

commd_t commd = {(int(*)())open,{(int)"/dev/s1", O_RDWR,
    asyncParityNone,asyncParityOdd,asyncStopBits1,
    asyncDataBits8,B9600}};
struct termios defaultAttr = { BRKINT|IGNCR|ICRNL|IGNPAR|IXON,
    OPOST,
    CLOCAL|CREAD|CS8,
    ECHO|ECHOE|ECHOK|ECHONL|ICANON|IEXTEN,
    {0x04, /* ^d EOF */
    0x00, /* ^@ EOL */
    0x08, /* ^h ERASE */
    0x03, /* ^c INTR */
    0x15, /* ^u KILL */
    0x1c, /* ^\ QUIT */
    0x1a, /* ^z SUSP */
    0x11, /* ^q START */
    0x13, /* ^s STOP */
    0, /* VMIN */
    0, /* VTIME */
    0,0,0,0,0}, /* spares */
    B9600, B9600}; /* speeds */

extern int async_init(driver_t *dsw, ...);

void main()
{
    int rc;
    int devhandle, fd1, fd2, fd3;

    rc=dev_io_init(16, 16);
    if (rc!=0) {
        s1dbprintf("error in dev_io_init, errno=%d\n", errno);
    }

    rc = driver_install(&devhandle, async_init);
    if (rc != 0) {
        s1dbprintf("error in driver_install, errno=%d\n", errno);
    }
    rc = device_install("/dev/s1", CHRTYPE, devhandle, 1, 128, 128);
    if (rc != 0) {
        s1dbprintf("error in device_install, errno=%d\n", errno);
    }

    rc = driver_install(&devhandle, tty_init);
```

```
if (rc != 0)
    s1dbprintf("%d driver_install(&devandle, tty_init, ...)\n", errno);
rc = device_install("/dev/tty0", DTYPE_TTY, devhandle,
                   &commd, NULL, &defaultAttr);

if (rc != 0)
    s1dbprintf(" %d device_install(\"/dev/tty0\"...)\n", errno);
fd1 = open("/dev/tty0", O_RDWR);
fd2 = open("/dev/tty0", O_RDWR);
fd3 = open("/dev/tty0", O_RDWR);

fs_init();    /* Initialize file system. */

shell(NULL);  /* Start the shell. */
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- **driver_install()**, p. 177
- **tty_init()**, p. 915

tzset()

Synopsis

```
#include <time.h>
void tzset( void );
```

Library

rtxLib.a

Description

tzset() uses the environment variable **TZ** to set the external variable **tzname**.

```
extern char *tzname[2] = {"std", "dst"};
```

The environment variable is defined as follows:

```
TZ=stdoffset[dst[offset],rule]
std = 3+ characters that cannot be 0-9, '+', '-', or ','
offset = hh[:mm[:ss]]
dst = 3+ characters that cannot be 0-9, '+', '-', or ','
rule = start[/time],end[/time]
start = Mm.n.d
end = Mm.n.d
M = literal 'M'
m = month (1-12)
n = Month week number (1-5, 5 = last week of the month)
d = day of the week (0-6, 0 = Sunday)
time = hh[:mm[:ss]]
hh = hours (0-23)
mm = minutes (0-59)
ss = seconds (0-59)
```

offset is the difference in local time from Universal Coordinated Time (UTC). A negative value indicates that the time zone is east of the prime meridian; a positive value indicates that the time zone is west of the prime meridian. *rule* defines when Daylight Saving Time goes into effect and when it ends. The items in brackets ([]) are optional; if they are not specified, a default value is used.

The default time zone for **TZ** is Eastern Standard Time in the United States:

```
TZ=EST5EDT,M4.1.0/02:00:00,M10.5.0/02:00:00
```

To change the default, use **setenv_np()**.

tzset() returns nothing.

Errors

None.

Example

See **fs_init()**, p. 231.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, §8.3.2
- **getenv()**, p. 262
- **setenv_np()**, p. 721

uname()

Synopsis

```
#include <sys/utsname.h>
int uname( struct utsname *name );
```

Library

rtxLib.a

Description

uname() stores information that identifies the current operating system; the information is in the structure pointed to by *name*.

The structure **utsname** is defined in the file **<sys/utsname.h>** and contains the following members:

sysname	Name of this implementation of the operating system.
nodename	Name of this node within an implementation-specified communications network.
release	Current release level of this implementation.
version	Current version level of this release.
machine	Name of the hardware type on which OS Open is running.

If successful, **uname()** returns a non-negative value. Otherwise, **uname()** returns **-1**.

Errors

None.

Example

See **shm_open()**, p. 734.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- IEEE Std 1003.1-1990, §4.4.1
- **setuname()**, p. 729

Synopsis

```
#include <stdio.h>

int ungetc( int c,
            FILE *stream );
```

Library

fsLib.a

Description

ungetc() pushes the character *c* onto the input stream pointed to by *stream*.

Only one **ungetc()** is guaranteed to be pushed onto the input stream, which must be open for reading. A subsequent read on the stream starts with *c*.

ungetc() cannot be used to push EOF on the stream. **ungetc()** fails if it is called immediately after a **fflush()** without an intervening read.

Characters placed on the stream by **ungetc()** are erased if **fseek()**, **fsetpos()**, **rewind()**, or **fflush()** are called before *c* is read from the stream.

ungetc() returns the integer argument *c* converted to an **unsigned char**. A return value of EOF indicates a failure to push *c* onto the stream.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

This example reads decimal characters from a file and converts them into an unsigned integer.

```
#include <stdio.h>

unsigned int read_number(FILE *file)
{
    int ch;
    unsigned int result = 0;
    while(((ch = getc(file)) != EOF) && (isdigit(ch)))
        result = (result * 10) + (ch - '0');
    /* put back the non-decimal character on the stream for future use */
    if(ch != EOF)
        ungetc(ch, file);
    return(result);
}
```

ungetc()

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.7.11
- **fflush()**, p. 197
- **fs_init()**, p. 231
- **fseek()**, p. 235
- **getc()**, p. 255
- **getchar()**, p. 257
- **putc()**, p. 609
- **putchar()**, p. 612

Synopsis

```
#include <unistd.h>

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

Library

devLib.a

Description

unlink() removes the link pointed to by *path* and decrements the link count of the referenced file.

When the link count of the file becomes 0 and no process has the file open, the file is deleted. If at least one process has the file is open when the last link is removed, deletion does not occur until all file references are closed.

If successful, **unlink()** returns 0. Otherwise, **unlink()** returns -1 and sets *errno*.

Errors

[EACCES]	Search permission is denied for a component of the path, or write permission is denied for the directory containing the file.
[EBUSY]	The directory cannot be unlinked because it is in use. This error requires a file system implementation that has a busy state for a directory.
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of <i>path</i> exceeds {PATH_MAX} .
[ENOENT]	The file named by <i>path</i> does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The file named by <i>path</i> is a directory. Implementations may optionally allow unlink() on directories.
[EROFS]	The requested link removal requires writing in a directory on a read-only file system.

Example

See **access()**, p. 55.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.5.1
- **link()**, p. 384

utime()

Synopsis

```
#include <types.h>
#include <utime.h>
int utime( const char *path, const struct utimbuf *newtimes );
```

Library

devLib.a

Description

utime() sets the access and modification times of the file pointed to by *path* to the time specified in the field pointed to by *newtimes*.

If the field pointed to by *newtimes* is NULL, the access and modifications times are set to the current time.

If successful, **utime()** returns 0. Otherwise, **utime()** returns -1 and sets *errno*.

Errors

[EACCES]	Search permission is denied for a component of the path.
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of <i>path</i> exceeds {PATH_MAX} .
[ENOENT]	<i>path</i> points to a file that does not exist or points to the empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The file permissions do not allow the times to be altered.
[EROFS]	The file pointed to by <i>path</i> is on a read-only file system

Example

The following example backdates a file to a date one year before the current date.

```
#include <types.h>
#include <time.h>
#include <utime.h>

int backdate(char *fname)
{
    int rc;
    struct tm s_tm;
    struct tm *p_tm;
    time_t l_time;

    struct utimbuf ut;
```

```

time(&l_time); /* get current time */
p_tm = gmtime(&l_time); /* convert to tm structure */
s_tm = *p_tm; /* assign to local structure */
s_tm.tm_year--;
l_time = mktime(&s_tm); /* Convert back to a time_t */
ut.actime = ut.modtime = l_time;
rc = utime(fname, &ut);
if (rc != 0) {
    perror("utime call");
}
return(rc);
}

```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §5.6.6

va_arg()

Synopsis

```
#include <stdarg.h>

var-type va_arg( va_list arg_ptr,
                var-type );
```

Library

cLib.a for High C/C++ compiler implementations. Otherwise, none. This is a macro defined in **<stdarg.h>**.

Description

va_arg() is used with **va_start()** and **va_end()** to access the arguments to a function when the function takes a fixed number of required arguments and a variable number of optional arguments.

va_arg() retrieves a value of the given *var-type* from the location given by *arg_ptr* and increases *arg_ptr* to point to the next argument in the list. **va_arg()** can retrieve arguments from the list any number of times within the function. *va_type* must be **int**, **long**, **short**, **char**, or **unsigned**, a **typedef** of one of these types, or a pointer.

va_arg() returns the current argument.

Errors

None.

Example

This example concatenates multiple strings into a common allocated string and returns the string's address.

```
#include <stdarg.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>

char *multcat(int numargs, ...)
{
    va_list arg;
    char *result;
    int i, siz = 0;

    /* get size required for resulting string */
    va_start(arg, numargs);
    for(i = 0; i < numargs; i++)
        siz += strlen(va_arg(arg, char *));
    va_end(arg);
```

```
/* allocate the memory for the resulting string */
result = (char *)malloc(siz + 1);
if(result == NULL)
    return(NULL);
/* concatenate the strings */
va_start(arg, numargs);
for(i = 0; i < numargs; i++)
    strcat(result, va_arg(arg, char *));
va_end(arg);
return(result);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.8.1.2
- **fprintf()**, p. 221
- **vfprintf()**, p. 930
- **vsprintf()**, p. 956

va_end()

Synopsis

```
#include <stdarg.h>

void va_end( va_list arg_ptr );
```

Library

cLib.a for High C/C++ compiler implementations. Otherwise, none. This is a macro defined in **<stdarg.h>**.

Description

va_end() is used with **va_start()** and **va_arg()** to access the arguments to a function when the function takes a fixed number of required arguments and a variable number of optional arguments.

You declare required arguments as ordinary parameters to the function and get to the arguments through the parameter names.

va_end() is used to clean up *arg_ptr*. To reset *arg_ptr* to the beginning of the variable list, use **va_start()**.

Errors

None.

Example

See **va_arg()**, p. 926.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.8.1.3
- **fprintf()**, p. 221
- **vfprintf()**, p. 930
- **vsprintf()**, p. 956

Synopsis

```
#include <stdarg.h>

void va_start( va_list arg_ptr,
               variable-name );
```

Library

cLib.a for High C/C++ compiler implementations. Otherwise, none. This is a macro defined in **<stdarg.h>**.

Description

va_start() is used with **va_arg()** and **va_end()** to access the arguments to a function when the function takes a fixed number of required arguments and a variable number of optional arguments.

You declare required arguments as ordinary parameters to the function and get to the arguments through the parameter names.

va_start() initializes **arg_ptr** for subsequent calls to **va_arg()** and **va_end()**. The argument **arg_ptr** must have a **va_list** type.

variable-name is the identifier of the rightmost named parameter in the parameter list. Use **va_start()** before **va_arg()**. Corresponding **va_start()** and **va_end()** macros must be in the same function.

Errors

None.

Example

See **va_arg()**, p. 926.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.8.1.1
- **fprintf()**, p. 221
- **vfprintf()**, p. 930
- **vsprintf()**, p. 956

fprintf()

Synopsis

```
#include <stdio.h>

int fprintf( FILE *stream,
             const char *format-string,
             va_list arg_ptr );
```

Library

fsLib.a

Description

fprintf() formats and writes a series of characters and values to the stream pointed to by *stream*.

fprintf() converts each entry in the argument list pointed to by *arg_ptr*, if any, and writes it to the stream according to the corresponding format specification in *format-string*.

format-string has the same form and function as *format-string* for **printf()**. See **printf()** on page 500 for a description of *format-string* and the argument list.

Both **fprintf()** and **fprintf()** can print a variable number of arguments. With **fprintf()**, the number of arguments is fixed at compile time, but with **fprintf()**, the number of arguments can be modified during program execution.

fprintf() formats and prints up to only 509 characters. The results are indeterminate if an attempt is made to print more than 509 characters.

fprintf() returns the number of characters written to the output stream or a negative value if an output error occurs.

Note: The file system must have been initialized by **fs_init()**.

Errors

None.

Example

The following example prints error messages to the standard error stream.

```
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

void fatal(char *format, ...)
{
    va_list arg;
    /* if errno is set display a system message */
```

```
if(errno)
    fprintf(stderr, "%s\n", strerror(errno));
/* if there is a user message, print that */
if(format != NULL)
{
    va_start(arg, format);
    vfprintf(stderr, format, arg);
    va_end(arg);
}
```

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.6.7
- **fs_init()**, p. 231
- **va_arg()**, p. 926
- **va_end()**, p. 928
- **va_start()**, p. 929

vm_addrealpages()

Synopsis

```
#include <vm.h>

int vm_addrealpages( tg_t thread_group, void * mem, size_t size,
                    void *req_mem, unsigned long attr );
```

Library

rtx.o

Description

vm_addrealpages() allocates *size* bytes of real memory in the virtual memory space specified by *mem*.

mem and *req_mem* address parameters must be page-aligned and *size* must be an integer multiple of the page size.

Allocated memory is associated with the thread group specified by *thread_group*.

req_mem specifies a real memory address for this operation.

attr can be one of **VM_ATTR_IOSPACE**, **VM_ATTR_ATADDR**, or **VM_ATTR_ALIASED**. **VM_ATTR_IOSPACE** specifies that the real memory is outside the region managed by the operating system. The application designer must manage pages to avoid duplicate mappings. **VM_ATTR_ALIASED** specifies that requested real memory has a valid mapping in the system and an additional virtual mapping is being requested. **VM_ATTR_ATADDR** allocates real memory from the region managed by the operating system, satisfying the request only if the specified real address range is available. If no attributes are specified, the real pages are selected from space managed by the operating system with no regard for real address.

The **VM_ATTR_IOSPACE**, **VM_ATTR_ATADDR**, and **VM_ATTR_ALIASED** attributes are mutually exclusive. If one of these attributes is specified, the *req_mem* parameter must also be provided. Otherwise *req_mem* is ignored.

When **VM_ATTR_IOSPACE**, **VM_ATTR_ATADDR**, or **VM_ATTR_ALIASED** is specified, the calling thread must be in **PTHREAD_PRIVILEGED_NP** state.

When adding real pages using the **VM_ATTR_ALIASED** attribute, alias only one page at a time, unless the real memory pointed to by *real_mem* is mapped contiguously.

If *thread_group* specifies the **PTHREAD_KERNEL_NP** thread group, the calling thread must be in the **PTHREAD_PRIVILEGED_NP** state.

If successful, **vm_addrealpages()** returns 0. Otherwise, **vm_addrealpages()** returns an error number.

Errors

[ENOMEM]	Insufficient pages to build page tables to add additional pages
----------	---

Example

The following example allocates a virtual memory page for the current thread group, backs it with a real page from the area managed by the Translation Manager, gets the starting address of the region, changes the protection and then removes and deallocates the pages.

```
#include <vm.h>
#include <pthread.h>

#define PAGESIZE 0x1000

int test()
{
    void * new_mem;
    int rc;
    size_t size;
    void * region_addr;
    unsigned long region_attr;

    rc = vm_allocate(pthread_tgself_np(), &new_mem, NULL, PAGESIZE, 1,
        VM_PROTECT_ALL);

    /* VM_PROTECT_ALL is provided in vm.h as a convenience and is equivalent
    to VM_PROTECT_READ|VM_PROTECT_WRITE|VM_PROTECT_EXECUTE
    */

    if (rc != 0)
        return(rc);
    rc = vm_addrealpages(pthread_tgself_np(), new_mem, PAGESIZE, NULL, 0);
    return(rc);
    rc = vm_region(pthread_tgself_np(), new_mem, &region_addr, &size,
        &region_attr);
    rc = vm_protect(pthread_tgself_np(), new_mem, PAGESIZE,
        VM_PROT_EXECUTE);
    rc = vm_remove_realpages(pthread_tgself_np(), new_mem, PAGESIZE);
    rc = vm_deallocate(pthread_tgself_np(), new_mem, PAGESIZE);

    return 0;
}
```

vm_addrealpages()

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes, unless <i>thread_group</i> specifies PTHREAD_KERNEL_NP

References

- **pthread_tgself_np()**, p. 606
- **vm_allocate()**, p. 935
- **vm_deallocate()**, p. 939
- **vm_protect()**, p. 941
- **vm_region()**, p. 947
- **vm_removeallpages()**, p. 949

Synopsis

```
#include <vm.h>

int vm_allocate( tg_t thread_group, void **new_mem, void *req_mem,
                size_t size, int find_flag, unsigned long attr);
```

Library

rtx.o

Description

vm_allocate() allocates virtual memory space for a thread group specified by the *thread_group* parameter.

A pointer to newly allocated virtual memory is returned in *new_mem*. *size* specifies, in bytes, the size of the allocated virtual memory.

When *find_flag* is set to 0, the system attempts to allocate virtual memory beginning at the address *req_mem*. Otherwise, the system selects the starting address of the virtual memory region.

The *req_mem* address parameter must be page-aligned and *size* must be an integer multiple of the page size.

The **PTHREAD_KERNEL_NP** thread group reserves virtual memory space from 0 through **VM_KERNEL_END**; the rest of the virtual memory space is available to other thread groups. If *thread_group* specifies the **PTHREAD_KERNEL_NP** thread group, the calling thread must be in **PTHREAD_PRIVILEGED_NP** state.

attr specifies the attribute of the new virtual memory region.

New regions can have the **VM_COPYON_WRITE** or **VM_EXTENDABLE** modifier applied to the attribute. **VM_COPYON_WRITE** specifies that a write operation to this region causes new real pages to be allocated; data is copied to the new pages before write access can continue. A write access to a **VM_COPYON_WRITE** region, made before real memory pages are allocated, causes a protection exception. **VM_EXTENDABLE** requests new real memory pages to be allocated in the new region whenever access to that region occurs.

attr can also specify **VM_PROTECT_READ**, **VM_PROTECT_WRITE**, **VM_PROTECT_EXECUTE**, **VM_ATTR_CACHEON**, **VM_ATTR_GUARDED**, **VM_ATTR_NOCOH**, **VM_ATTR_CWRITETH**. If **VM_COPYON_WRITE** is specified, the **VM_PROTECT_WRITE** attribute must not be used.

For an allocated region to be accessible, one of the **VM_PROTECT_READ**, **VM_PROTECT_WRITE**, or **VM_PROTECT_EXECUTE** attributes must be specified.

If successful, **vm_allocate()** returns 0. Otherwise, **vm_allocate()** returns an error number.

vm_allocate()

Errors

[EINVAL]	The <i>req_mem</i> , <i>size</i> , or <i>attr</i> parameter is invalid.
[ENOENT]	The <i>thread_group</i> parameter is invalid.
[ENOMEM]	No real memory is available for the request.
[ENOSPC]	No virtual memory space is available.
[EPERM]	Operation not permitted.

Example

Refer to **vm_addrealpages()**, p. 932.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes, unless <i>thread_group</i> specifies PTHREAD_KERNEL_NP

References

- **vm_addrealpages()**, p. 932
- **vm_deallocate()**, p. 939
- **vm_protect()**, p. 941
- **vm_region()**, p. 947
- **vm_removeallpages()**, p. 949

Synopsis

```
#include <vm.h>

int vm_copy( tg_t *thread_group, void *source, size_t size,
             void *destination);
```

Library

rtx.o

Description

vm_copy() copies *size* bytes from *source* address to *destination* address in the thread group specified by *thread_group*.

source and *destination* address parameters have to be page aligned and *size* must be an integer multiple of the page size.

If *thread_group* specifies the **PTHREAD_KERNEL_NP** thread group, the calling thread must be in **PTHREAD_PRIVILEGED_NP** state.

If successful, **vm_copy()** returns 0. Otherwise, **vm_copy()** returns an error number.

Errors

[EINVAL]	<i>source</i> , <i>destination</i> , or <i>size</i> is invalid.
[ENOENT]	<i>thread_group</i> is invalid.
[EPERM]	Operation not permitted.

Example

In the following example, *copy_it()* allocates and adds a real page. *copy_it()* will then take one page of data from the virtual address 0x10001000 for thread group *thread_g* and copy it into the new page pointed to by *new_mem*.

```
#include <vm.h>
#include <pthread.h>

#define PAGESIZE 0x1000

int copy_it(tg_t * thread_g)
{
    void * new_mem;
    int rc;
    size_t size;
    void * region_addr;
    unsigned long region_attr;

    rc = vm_allocate(pthread_tgself_np(), &new_mem, NULL, PAGESIZE, 1,
                     VM_PROTECT_ALL);
```

vm_copy()

```
/* VM_PROTECT_ALL is provided in vm.h as a convenience and is equivalent
to VM_PROTECT_READ|VM_PROTECT_WRITE|VM_PROTECT_EXECUTE
*/
if (rc != 0)
    return(rc);
rc = vm_addrealpages(pthread_tgself_np(), new_mem, PAGE_SIZE, NULL, 0);
if (rc != 0)
    return(rc);
rc = vm_copy(thread, 0x10001000, PAGE_SIZE, new_mem);
return rc;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes, unless <i>thread_group</i> specifies PTHREAD_KERNEL_NP

References

- **vm_allocate()**, p. 935
- **vm_deallocate()**, p. 939
- **vm_protect()**, p. 941
- **vm_read()**, p. 945
- **vm_region()**, p. 947
- **vm_write()**, p. 953

Synopsis

```
#include <vm.h>

int vm_deallocate( tg_t thread_group, void *mem, size_t size);
```

Library

rtx.o

Description

vm_deallocate() deallocates virtual memory space for a thread group specified by the *thread_group* parameter.

mem is a pointer to the memory to be deallocated. *size* specifies the size to be deallocated.

The *mem* address must be page-aligned and *size* must be an integer multiple of the page size.

Any real memory pages obtained from the operating system and mapped for the virtual memory that is being deallocated are returned to the free page pool.

If *thread_group* specifies the **PTHREAD_KERNEL_NP** thread group, the calling thread must be in the **PTHREAD_PRIVILEGED_NP** state.

If successful, **vm_deallocate()** returns 0. Otherwise, **vm_deallocate()** returns an error number.

Errors

[EDEADLK]	<i>mem</i> or <i>size</i> is invalid.
[EINVAL]	<i>mem</i> or <i>size</i> is invalid.
[ENOENT]	<i>thread_group</i> is invalid.
[ENOMEM]	No real memory is available for the request.
[EPERM]	Operation not permitted.

Example

Refer to **vm_addrealpages()**, p. 932.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes, unless <i>thread_group</i> specifies PTHREAD_KERNEL_NP

vm_deallocate()

References

- [vm_addrealpages\(\)](#), p. 932
- [vm_allocate\(\)](#), p. 935
- [vm_deallocate\(\)](#), p. 939
- [vm_protect\(\)](#), p. 941
- [vm_region\(\)](#), p. 947
- [vm_remove_realpages\(\)](#), p. 949

Synopsis

```
#include <vm.h>

int vm_protect( tg_t thread_group, void *mem, size_t size,
               unsigned long protection);
```

Library

rtx.o

Description

vm_protect() sets the protection attributes for the virtual memory range specified by *mem* and *size*.

protection can specify the following attributes: **VM_PROTECT_READ**, **VM_PROTECT_WRITE**, **VM_PROTECT_EXECUTE**, **VM_PROTECT_ALL**, **VM_PROTECT_NONE**, **VM_CACHEON**, **VM_ATTR_GUARDED**, **VM_ATTR_NOCOH** or **VM_ATTR_CWRITETH**.

The *mem* address must be page-aligned and *size* must be an integer multiple of the page size.

If *thread_group* specifies the **PTHREAD_KERNEL_NP** thread group, the calling thread must be in the **PTHREAD_PRIVILEGED_NP** state.

If successful, **vm_protect()** returns 0. Otherwise, **vm_protect()** returns an error number.

Errors

[EINVAL]	<i>mem</i> or <i>size</i> is invalid.
[ENOENT]	<i>thread_group</i> is invalid.
[ENOMEM]	No real memory is available for the request.
[EPERM]	Operation not permitted.

Example

Refer to **vm_addrealpages()**, p. 932

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes, unless <i>thread_group</i> specifies PTHREAD_KERNEL_NP

vm_protect()

References

- `vm_allocate()`, p. 935
- `vm_copy()`, p. 937
- `vm_deallocate()`, p. 939
- `vm_read()`, p. 945
- `vm_region()`, p. 947
- `vm_write()`, p. 953

Synopsis

```
#include <vm.h>

int vm_query( struct vm_stat *tms);

struct vm_stats {
    unsigned int num_real_pages;
    unsigned int free_real_pages;
    unsigned int rp_high_water;
    unsigned int tlb_miss_count;
    unsigned int page_fault_count;
    unsigned int protection_excp_count;
    unsigned int free_trans_ids;
};
```

Library

rtx.o

Description

vm_query() returns Translation Manager statistics to a calling application.

- *num_real_pages*: The number of real pages being managed by the Translation Manager, not counting pages for which **vm_addrealpages()** supplied real addresses.
- *free_real_pages*: The number of pages in the page pool available for allocation by **vm_addrealpages()**.
- *rp_high_water*: The maximum number of real pages allocated since the last call to **vm_query()**. The call resets this value to the current page utilization. Calling **vm_query()** periodically provides a picture of the real page utilization over time can be determined.
- *tlb_miss_count*: The number of times a TLB miss handler was called to bring a page entry into the TLB. This field is reset after the call to **vm_query()**.
- *page_fault_count*: The number of times the Translation Manager could not resolve a virtual page reference. This field is reset after the call to **vm_query()**.
- *protection_excp_count*: The number of times a virtual mapping existed for a page, but an access violation occurred, resulting in a call to **vm_resolve()**. An example is a write operation to a page marked Read Only. This field is reset after the call to **vm_query()**.
- *free_trans_ids*: The number of available translation IDs. Each new thread group requires a unique, unused translation ID.

vm_query() returns 0 to the calling application (there are no errors).

vm_query()

Errors

None.

Example

The following example prints out a message if less than three free pages of real storage are available.

```
#include <vm.h>
#include <pthread.h>

int test()
{
    struct vm_stat vm_s;
    int rc;
    rc = vm_query(&vm_s);
    if (vm_s.free_real_pages < 3)
        printf(" There are only %d pages free\n",vm_s.free_real_pages);
    return 0;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes

Synopsis

```
#include <vm.h>

int vm_read( tg_t thread_group, void *source, size_t size,
             void **destination);
```

Library

rtx.o

Description

vm_read() transfers *size* bytes at *source* address from *thread_group* to the current thread group.

New real pages are acquired from the system pool and placed at *destination* address. Data is copied from *thread_group* at address *source* into this new area. The allocated region has the **VM_PROTECT_ALL** and **VM_ATTR_CACHE_ON** attributes.

The *source* address must be page-aligned and *size* must be an integer multiple of the page size.

The calling thread must be in the **PTHREAD_PRIVILEGED_NP** state.

If successful, **vm_read()** returns 0. Otherwise, **vm_read()** returns an error number.

Errors

[EINVAL]	<i>source</i> or <i>size</i> is invalid.
[ENOENT]	<i>thread_group</i> is invalid.
[ENOMEM]	No real pages exist for the request.
[EPERM]	Operation not permitted.

Example

In the following example, *read_it()* determines whether it is running in the kernel thread group. If so, it reads a page of data from the *other_tg* thread group virtual address 0x10001000. The data is read into a newly allocated data area pointed to by *other_data*.

```
#include <vm.h>
#include <pthread.h>

int read_it(tg_t other_tg)
{
    void * other_data;
    int rc;
```

vm_read()

```
/* verify that we are in the kernel thread group */
if(pthread_tgself_np() != pthread_tggetkernel_np())
    return -1;
rc = vm_read(other_tg, 0x10001000, PAGE_SIZE, &other_data);
return rc;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	No

References

- **vm_allocate()**, p. 935
- **vm_copy()**, p. 937
- **vm_deallocate()**, p. 939
- **vm_protect()**, p. 941
- **vm_region()**, p. 947
- **vm_write()**, p. 953

Synopsis

```
#include <vm.h>

int vm_region( tg_t thread_group, void *mem, void **region_addr,
               size_t *region_size, unsigned long *region_attributes);
```

Library

rtx.o

Description

vm_region() searches, starting at address *mem*, for the next virtual memory region. A virtual memory region is a virtual memory mapping with identical attributes, including whether a virtual page is backed by a real page.

If a region is found, its starting address, size and attribute values are returned in the *region_addr*, *region_size*, and *region_attributes* variables, respectively.

Attributes can have one or more of the following values: **VM_PROTECT**, **VM_PRIVATE**, **VM_COPYON_WRITE**, **VM_EXTENDABLE**, **VM_ATTR_CACHEON**, **VM_ATTR_GUARDED**, **VM_ATTR_NOCOH**, **VM_ATTR_CWRITETH**, **VM_ATTR_IOSPACE**, **VM_ATTR_ATADDR**, **VM_ATTR_ALIASED**, and **VM_RPAGE_MAPPED**.

VM_RPAGE_MAPPED indicates that real memory pages are assigned to the virtual memory region.

If *thread_group* specifies the **PTHREAD_KERNEL_NP** thread group, the thread executing this call must be in the **PTHREAD_PRIVILEGED_NP** state.

If successful, **vm_region()** returns 0. Otherwise, **vm_region()** returns an error number.

Errors

[EINVAL]	No virtual memory region was found.
[ENOENT]	<i>thread_group</i> is invalid.
[EPERM]	Operation not permitted.

Example

Refer to **vm_addrealpages()**, p. 932.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes

vm_region()

References

- `vm_allocate()`, p. 935
- `vm_copy()`, p. 937
- `vm_deallocate()`, p. 939
- `vm_protect()`, p. 941
- `vm_read()`, p. 945
- `vm_write()`, p. 953

Synopsis

```
#include <vm.h>
```

```
int vm_remove_realpages( tg_t thread_group, void *mem, size_t size);
```

Library

rtx.o

Description

vm_remove_realpages() removes mappings for *size* bytes of real memory in the virtual memory space specified by *mem*.

If the real pages were initially obtained from the operating system, they are returned to the free page pool.

thread_group specifies the thread group associated with that virtual address.

The *mem* address parameter has to be page aligned and *size* must be an integer multiple of the page size.

If *thread_group* specifies the **PTHREAD_KERNEL_NP** thread group, the calling thread must be in the **PTHREAD_PRIVILEGED_NP** state.

If successful, **vm_remove_realpages()** returns 0. Otherwise, **vm_remove_realpages()** returns an error number.

Errors

[EINVAL]	<i>mem</i> or <i>size</i> is invalid.
[ENOENT]	<i>thread_group</i> is invalid.
[ENOMEM]	No real memory is available for the request.
[EPERM]	Operation not permitted.

Example

Refer to **vm_add_realpages()**, p. 932.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	Yes, unless <i>thread_group</i> specifies PTHREAD_KERNEL_NP

vm_remove_realpages()

References

- [vm_addrealpages\(\)](#), p. 932
- [vm_allocate\(\)](#), p. 935
- [vm_copy\(\)](#), p. 937
- [vm_deallocate\(\)](#), p. 939
- [vm_protect\(\)](#), p. 941
- [vm_region\(\)](#), p. 947

Synopsis

```
#include <vm.h>

int vm_translate( tg_t thread_group, void *address, void **raddress);
```

Library

rtx.o

Description

vm_translate() returns, in the variable pointed to by *raddress*, the real memory address corresponding to the virtual address *address* in the thread group specified by *thread_group*.

If *thread_group* specifies the **PTHREAD_KERNEL_NP** thread group, the calling thread must be in the **PTHREAD_PRIVILEGED_NP** state.

If successful, **vm_translate()** returns 0. Otherwise, **vm_translate()** returns an error number.

Errors

[EINVAL]	The <i>address</i> does not have a real page mapped to it.
[ENOENT]	<i>thread_group</i> is invalid.

Example

In the following example, *test()* allocates and adds a real page, then translates the virtual address of the new page to a real address and prints the results.

```
#include <vm.h>
#include <pthread.h>

#define PAGESIZE 0x1000

int test()
{
    void * new_mem, real_addr;
    int rc;
    size_t size;
    void * region_addr;
    unsigned long region_attr;
    rc = vm_allocate(pthread_tgself_np(), &new_mem, NULL, PAGESIZE, 1,
        VM_PROTECT_ALL);
    if (rc != 0)
        return(rc);
    rc = vm_addrealpages(pthread_tgself_np(), new_mem, PAGESIZE, NULL, 0);
    if (rc != 0)
        return(rc);
```

vm_translate()

```
rc = vm_translate(pthread_tgself_np(), new_mem, &real_addr);
printf(" for thread group %x virtual address %x = %x\n", pthread_tgself_np(),
new_mem, real_addr);
return rc;
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from App. Thread	Yes, unless <i>thread_group</i> specifies PTHREAD_KERNEL_NP

References

- **vm_allocate()**, p. 935
- **vm_copy()**, p. 937
- **vm_deallocate()**, p. 939
- **vm_protect()**, p. 941
- **vm_read()**, p. 945
- **vm_write()**, p. 953

Synopsis

```
#include <vm.h>

int vm_write( tg_t thread_group, void *destination, void *source,
              size_t size);
```

Library

rtx.o

Description

vm_write() transfers *size* bytes from *source* address in the current thread group to *destination* address in the thread group specified by *thread_group*. The destination region must already exist and be writable.

The *source* and *destination* addresses must be page-aligned and *size* must be an integer multiple of the page size.

The calling thread must be in the **PTHREAD_PRIVILEGED_NP** state.

If successful, **vm_write()** returns 0. Otherwise, **vm_write()** returns an error number.

Errors

[EINVAL]	<i>source</i> , <i>destination</i> , or <i>size</i> is invalid.
[ENOENT]	<i>thread_group</i> is invalid.
[EPERM]	Operation not permitted.

Example

In the following example, *write_it()* determines whether it is running in the kernel thread group. If so, it takes a page of data from the area pointed to by *other_data* and writes the data to the thread group *target_group* at virtual address 0x10001000.

```
#include <vm.h>
#include <pthread.h>

int write_it(void * other_data, tg_t target_group)
{
    int rc;
    /* verify that we are in the kernel thread group */
    if(pthread_tgself_np() != pthread_tggetkernel_np())
        return -1;
    rc = vm_write(target_group, (void *)0x10001000, PAGESIZE, other_data);
    return rc;
}
```

vm_write()

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from App. Thread	No

References

- **vm_allocate()**, p. 935
- **vm_copy()**, p. 937
- **vm_deallocate()**, p. 939
- **vm_protect()**, p. 941
- **vm_read()**, p. 945
- **vm_region()**, p. 947

Synopsis

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

int vprintf( const char *format, va_list arg );
```

Library

fsLib.a

Description

vprintf() is equivalent to **printf()**, with the variable argument list replaced by *arg*.

arg is initialized by **va_start()** (and possibly subsequent **va_arg()** calls). **vprintf()** does not invoke **va_end()**.

If successful, **vprintf()** returns the number of characters sent. Otherwise, **vprintf()** returns a negative number.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- ANSI X3.159-1989, §4.9.10.4
- **printf()**, p. 500
- **va_arg()**, p. 926
- **va_start()**, p. 929

vsprintf()

Synopsis

```
#include <stdio.h>

int vsprintf( char *buffer,
              const char *format-string,
              va_list arg_ptr );
```

Library

cLib.a

Description

vsprintf() formats and stores a series of characters and values in *buffer*, which is an array of characters.

vsprintf() converts each entry in the argument list pointed to by *arg_ptr*, if any, and writes to the buffer according to the corresponding format specification in *format-string*.

format-string has the same form and function as *format-string* for **printf()**. See **printf()** on page 500 for a description of *format-string* and the argument list.

Both **sprintf()** and **vsprintf()** can print a variable number of arguments. With **sprintf()**, the number of arguments is fixed at compile time, but with **vsprintf()**, the number of arguments can be modified during program execution.

vsprintf() returns the number of characters written in the array, not counting the ending NULL character.

Errors

None.

Example

This example formats an array of numbers into a string. A flag tells whether to format the numbers as decimal, hexadecimal, or octal.

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

#define HEX 1
#define OCT 2
#define DEC 3

int numbers[7];

char *format_array(int flag)
{
    char *buffer;
```

```
int *ptr;

/* malloc storage for the buffer */
buffer = (char *)malloc(120);
/* Set ptr to point to the address of an element one before the */
/* beginning of the array so the macro va_arg() in vsprintf() */
/* will read the first element in the array. */
ptr = numbers;
/* format the array according to the flag */
if(flag == HEX)
    vsprintf(buffer, "%#x, %#x, %#x, %#x, %#x, %#x, %#x\n",
              (va_list)ptr);
else if(flag == OCT)
    vsprintf(buffer, "%#o, %#o, %#o, %#o, %#o, %#o, %#o\n",
              (va_list)ptr);
else
    vsprintf(buffer, "%d, %d, %d, %d, %d, %d, %d\n",
              (va_list)ptr);
return(buffer);
}
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.9.6.9
- **va_arg()**, p. 926
- **va_end()**, p. 928
- **va_start()**, p. 929

wctomb()

Synopsis

```
#include <stdlib.h>

int wctomb( char *s, wchar_t wchar );
```

Library

cLib.a

Description

wctomb() returns 1.

OS Open does not support multibyte characters; **wctomb()** is provided for ANSI C compatibility.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.7.3

Synopsis

```
#include <stdlib.h>
size_t wcstombs( char *s, wchar_t *pcws, size_t n );
```

Library

cLib.a

Description

wcstombs() returns the value of *n*.
OS Open does not support multibyte characters; **wcstombs()** is provided for ANSI C compatibility.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

- ANSI X3.159-1989, §4.10.8.2

write()

Synopsis

```
#include <unistd.h>

ssize_t write( int filedes, const void *buffer, size_t nbytes );
```

Library

devLib.a

Description

write() writes *nbytes* bytes from *buffer* to the file associated with the descriptor *filedes*.

If *nbytes* is 0, **write()** returns 0 with no other results. If the file is capable of seeking, **write()** starts from and updates the file offset. Otherwise, the value of the file offset is undefined and **write()** starts from the current position. If the **O_APPEND** flag of the file status flag is set (and the device supports it), the file offset is set to the end of the file before each write.

write() returns the number of bytes actually written to the file. This number may be less than *nbytes* if the space in the file did not permit the writing of *nbytes*, a blocking **write()** request was interrupted by a signal, or a special file has insufficient space to immediately write *nbytes* bytes.

When attempting to write to a file that supports nonblocking writes and there is insufficient space, **write()** returns -1 with *errno* set to [EAGAIN] if **O_NONBLOCK** was specified on **open()**. Otherwise, **write()** blocks until the data can be accepted.

If successful, **write()** returns the number of bytes transferred from the buffer to the file. Otherwise, **write()** returns -1 and sets *errno*.

Errors

[EBADF]	<i>filedes</i> is not a valid file descriptor.
[EAGAIN]	Nonblocking writes are supported, and the O_NONBLOCK flag is set for this file descriptor. write() would block until the data is accepted, but there is insufficient space in the device.
[EINTR]	An unblocked signal is pending on OS Open or the thread.

Example

See **read()**, p. 633.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- IEEE Std 1003.1-1990, §6.4.2
- **open()**, p. 478

xdr_accepted_reply()

Synopsis

```
#include <rpc/rpc.h>
```

```
bool_t xdr_accepted_reply( XDR *xdrs, struct accepted_reply *ar );
```

Library

rpcLib.a

Description

xdr_accepted_reply() encodes an RPC reply message.

xdr_accepted_reply() generates message replies, similar to RPC message replies, without using the RPC program.

xdrs points to the XDR stream handle; *ar* specifies the address of the structure that contains the RPC reply.

If successful, **xdr_accepted_reply()** returns 1. Otherwise, **xdr_accepted_reply()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>

bool_t xdr_array( XDR *xdrs, char **arrp, u_int *sizep, u_int maxsize,
                  u_int elsize, xdrproc_t elproc );
```

Library

rpcLib.a

Description

xdr_array(), an XDR filter primitive, translates between a variable-length array and its external representation.

xdr_array() is called to encode or decode each array element.

xdrs points to the XDR stream handle.

arrp specifies the address of the pointer to the array. If *arrp* is NULL when the array is deserialized, XDR allocates an array of the appropriate size and sets *arrp* to the newly allocated array.

sizep specifies the address of the array element count; the element count cannot exceed the value of *maxsize*.

maxsize specifies the maximum number of array elements.

elsize specifies the size, in bytes, of the array elements.

elproc specifies the XDR filter that translates between the C form of the array elements and their external representation.

If successful, **xdr_array()** subroutine returns 1. Otherwise, **xdr_array()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_authunix_params()

Synopsis

```
#include <rpc/rpc.h>
```

```
bool_t xdr_authunix_params( XDR *xdrs, struct authunix_params *app );
```

Library

rpcLib.a

Description

xdr_authunix_params() generates UNIX credentials.

xdrs points to the XDR stream handle; *app* points to a structure containing the UNIX authentication credentials.

If successful, **xdr_authunix_params()** returns 1. Otherwise, **xdr_authunix_params()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_bool( XDR *xdrs, bool_t *bp );
```

Library

rpcLib.a

Description

xdr_bool(), an XDR filter primitive, translates between a C **bool_t** type and its external representation.

xdrs points to the XDR stream handle; *bp* points to the **bool_t** value.

If successful, **xdr_bool()** returns 1. Otherwise, **xdr_bool()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_bytes()

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t xdr_bytes( XDR *xdrs, char **sp, u_int *sizep, u_int maxsize );
```

Library

rpcLib.a

Description

xdr_bytes(), an XDR filter primitive, translates between a C counted byte array and its external representation.

xdr_bytes() operates on a subset of generic arrays; each array element contains one byte.

xdrs points to the XDR stream handle; *sp* points to the address of the array.

The length of the byte array is explicitly located in the unsigned integer pointed to by *sizep*, which contains the element count of the array. The element count cannot exceed *maxsize*, which specifies the maximum number of bytes allowed when XDR encodes or decodes messages.

The byte sequence is not terminated by a NULL character. The external representation of the bytes is the same as their internal representation.

If successful, **xdr_bytes()** returns 1. Otherwise, **xdr_bytes()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_callhdr( XDR *xdrs, struct rpc_msg *chdr );
```

Library

rpcLib.a

Description

xdr_callhdr() generates call headers similar to RPC call headers.

xdrs points to the XDR stream handle; *chdr* points to the structure containing the header for the call message.

If successful, **xdr_callhdr()** returns 1. Otherwise, **xdr_callhdr()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_callmsg()

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_callmsg( XDR *xdrs, struct rpc_msg *cmmsg );
```

Library

rpcLib.a

Description

xdr_callmsg() generates messages similar to RPC messages.

xdrs points to the XDR stream handle; *cmmsg* points to the structure containing the call message text.

If successful, **xdr_callmsg()** returns 1. Otherwise, **xdr_callmsg()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_char( XDR *xdrs, char *cp );
```

Library

rpcLib.a

Description

xdr_char(), an XDR filter primitive, translates between the C **char** pointed to by *cp* and its external representation.

Encoded characters, which are not packed, each occupy four bytes.

The **xdr_bytes()**, **xdr_opaque()**, or **xdr_string()** functions are provided for translating character arrays.

xdrs points to the XDR stream handle.

If successful, **xdr_char()** returns 1. Otherwise, **xdr_char()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_destroy(), XDR_DESTROY()

Synopsis

```
#include <rpc/rpc.h>
void xdr_destroy( XDR *xdrs );
```

Library

None. This is a macro defined in `<rpc/xdr.h>`.

Description

xdr_destroy(), also called **XDR_DESTROY()**, invokes the destroy function associated with the XDR stream pointed to by *xdrs* and frees the private data structures allocated to the stream.

The XDR stream handle is undefined after it is destroyed.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- `rpc_thread_init()`, p. 676

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_double( XDR *xdrs, double *dp );
```

Library

rpcLib.a

Description

xdr_double(), an XDR filter primitive, translates between a C **double** and its external representation.

dp points to the double-precision number; *xdrs* points to the XDR stream handle.

If successful, **xdr_double()** returns 1. Otherwise, **xdr_double()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_enum()

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_enum( XDR *xdrs, enum_t *ep );
```

Library

rpcLib.a

Description

xdr_enum(), an XDR filter primitive, translates between a C **enum** and its external representation.

xdrs points to the XDR stream handle; *ep* points to the enumeration data.

If successful, **xdr_enum()** returns 1. Otherwise, **xdr_enum()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_float( XDR *xdrs, float *fp );
```

Library

rpcLib.a

Description

xdr_float(), an XDR filter primitive, translates between a C **float** (normalized single floating-point number) and its external representation.

xdrs points to the XDR stream handle; *fp* points to the **float**.

xdr_float() requires a math coprocessor in CPQ.

If successful, **xdr_float()** returns 1. Otherwise, **xdr_float()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_free()

Synopsis

```
#include <rpc/rpc.h>
void xdr_free( xdrproc_t proc, char *objp );
```

Library

rpcLib.a

Description

xdr_free() provides a generic storage deallocation routine.

proc specifies the XDR routine for the object being freed; *fp* points to the object.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
u_int xdr_getpos( XDR *xdrs );
```

Library

None. This is a macro defined in `<rpc/xdr.h>`.

Description

xdr_getpos(), also called **XDR_GETPOS()**, invokes the get-position routine associated with the XDR stream handle pointed to by *xdrs*.

xdr_getpos() returns an unsigned integer describing the current position in the XDR stream. In some XDR streams, **xdr_getpos()** returns `-1`, even though the value has no meaning.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **xdr_setpos()**, **XDR_SETPOS()**, p. 987

xdr_inline(), XDR_INLINE()

Synopsis

```
#include <rpc/rpc.h>
long *xdr_inline( XDR *xdrs, int len );
```

Library

None. This is a macro defined in `<rpc/xdr.h>`.

Description

xdr_inline(), also called **XDR_INLINE()**, invokes the in-line routine associated with the XDR stream handle pointed to by *xdrs*.

xdr_inline() returns a pointer to a contiguous segment of the stream buffer.

len specifies the size of the buffer in bytes. The buffer can be used for any purpose but it is not data-portable.

xdr_inline() may return a NULL if the macro cannot return a buffer segment of the requested size.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- `rpc_thread_init()`, p. 676

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_int( XDR *xdrs, int *ip );
```

Library

rpcLib.a

Description

xdr_int(), an XDR filter primitive, translates between a C **int** and its external representation.

xdrs points to the XDR stream handle; *ip* points to the integer.

If successful, **xdr_int()** returns 1. Otherwise, **xdr_int()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_long()

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_long( XDR *xdrs, long *lp );
```

Library

rpcLib.a

Description

xdr_long(), an XDR filter primitive, translates between a C **long** and its external representation.

xdrs points to the XDR stream handle; *lp* specifies the address of the long integer.

If successful, **xdr_long()** returns 1. Otherwise, **xdr_long()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t xdr_opaque( XDR *xdrs, char *cp, u_int cnt );
```

Library

rpcLib.a

Description

xdr_opaque(), an XDR filter primitive, translates between fixed-size opaque data and its external representation.

xdrs points to the XDR stream handle; *cp* specifies the address of the opaque object.

cnt specifies the size, in bytes, of the object. By definition, data in the opaque object is not machine-portable.

If successful, **xdr_opaque()** returns 1. Otherwise, **xdr_opaque()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_opaque_auth()

Synopsis

```
#include <rpc/rpc.h>

bool_t xdr_opaque_auth( XDR *xdrs, struct opaque_auth *ap );
```

Library

rpcLib.a

Description

xdr_opaque_auth() generates an RPC authentication information message similar to an RPC authentication information message.

xdrs points to the XDR stream handle; *ap* points to the structure containing the authentication information.

If successful, **xdr_opaque_auth()** subroutine returns 1. Otherwise, **xdr_opaque_auth()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_pmap( XDR *xdrs, struct pmap *regs );
```

Library

rpcLib.a

Description

xdr_pmap() describes parameters for portmap procedures and generates portmap parameters without using the portmap interface.

xdrs points to the XDR stream handle; *regs* points to the buffer or register where the portmap daemon stores information.

If successful, **xdr_pmap()** returns 1. Otherwise, **xdr_pmap()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_pmaplist()

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_pmaplist( XDR *xdrs, struct pmaplist **rp );
```

Library

rpcLib.a

Description

xdr_pmaplist() describes a list of port mappings externally and generates the port mappings to RPC ports without using the portmap interface.

xdrs points to the XDR stream handle; *rp* points to the structure containing the portmap listings.

If successful, **xdr_pmaplist()** returns 1. Otherwise, **xdr_pmaplist()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t xdr_pointer( XDR *xdrs, char **objpp, u_int objsize,
                   xdrproc_t xdrobj );
```

Library

rpcLib.a

Description

xdr_pointer() translates between a C pointer and its external representation.

xdr_pointer() can represent recursive data structures, such as binary trees or linked lists.

xdrs points to the XDR stream handle.

objpp points to the pointer of the data structure; *objsize* specifies its size.

xdrobj specifies the XDR filter for the object.

If successful, **xdr_pointer()** returns 1. Otherwise, **xdr_pointer()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_reference()

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t xdr_reference( XDR *xdrs, char **pp, u_int size, xdrproc_t proc );
```

Library

rpcLib.a

Description

xdr_reference(), an XDR filter primitive, translates between a C pointer and its external representation.

xdr_pointer() allows the serializing, deserializing, and freeing of a pointer, within a structure, referenced by another structure.

xdr_reference() does not attach a special meaning to a NULL pointer during serialization. Attempting to pass the address of a NULL pointer can cause a storage error.

Data must be described using a two-armed discriminated union. One arm is used if the pointer is valid; the other arm is used if the pointer is NULL.

xdrs points to the XDR stream handle.

pp specifies the address of the pointer to the structure; *size* specifies its size, in bytes. When decoding data, XDR allocates storage if the pointer is NULL.

proc filters the structure between its C form and its external representation. This parameter is the XDR procedure describing the structure.

If successful, **xdr_reference()** returns 1. Otherwise, **xdr_reference()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_rejected_reply( XDR *xdrs, struct rejected_reply *rr );
```

Library

rpcLib.a

Description

xdr_rejected_reply() describes RPC message rejection replies.

xdr_rejected_reply() can generate rejection replies, similar to RPC rejection replies, without using the RPC program.

xdrs points to the XDR stream handle; *rr* points to the structure containing the rejected reply.

If successful, **xdr_rejected_reply()** returns 1. Otherwise, **xdr_rejected_reply()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_replymsg()

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_replymsg( XDR *xdrs, struct rpc_msg *rmsg );
```

Library

rpcLib.a

Description

xdr_replymsg() describes RPC message replies.

xdr_replymsg() generates message replies, similar to RPC message replies, without using the RPC program.

xdrs points to the XDR stream handle; *rmsg* points to the structure containing the parameters of the reply message.

If successful, **xdr_replymsg()** returns 1. Otherwise, **xdr_replymsg()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t xdr_setpos( XDR *xdrs, u_int pos );
```

Library

None. This is a macro defined in `<rpc/xdr.h>`.

Description

xdr_setpos(), also called **XDR_SETPOS()** invokes the set-position routine associated with the XDR stream pointed to by *xdrs* parameter.

The new position setting is obtained using **xdr_getpos()**, which returns FALSE if the set position is impossible, or if the requested position is out of bounds.

A position cannot be set in some XDR streams. Trying to set position in such streams causes **xdr_setpos()** to fail. **xdr_setpos()** also fails if the requested a position is not within the boundary of the stream.

xdrs points to the XDR stream handle; *pos* specifies a position value obtained using **xdr_getpos()**.

If successful, **xdr_setpos()** returns 1. Otherwise, **xdr_setpos()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **xdr_getpos()**, **XDR_GETPOS()**, p. 975

xdr_short()

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_short( XDR *xdrs, short *sp );
```

Library

rpcLib.a

Description

xdr_short(), an XDR filter primitive, translates between a C short integer and its external representation.

xdrs points to the XDR stream handle; *sp* specifies the address of the short integer.

If successful, **xdr_short()** returns 1. Otherwise, **xdr_short()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t xdr_string( XDR *xdrs, char **sp, u_int maxsize );
```

Library

rpcLib.a

Description

xdr_string(), an XDR filter primitive, translates between a C string and its external representation.

xdrs points to the XDR stream handle; *sp* specifies the address of the pointer to the string.

maxsize specifies the maximum string length allowed during encoding and decoding. This value is set in a protocol.

If successful, **xdr_string()** returns 1. Otherwise, **xdr_string()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_u_char()

Synopsis

```
#include <rpc/rpc.h>
bool_t xdr_u_char( XDR *xdrs, char *ucp );
```

Library

rpcLib.a

Description

xdr_u_char(), an XDR filter primitive, translates between a C **u_char** and its external representation.

xdrs points to the XDR stream handle; *ucp* points to the unsigned character.

If successful, **xdr_u_char()** returns 1. Otherwise, **xdr_u_char()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t xdr_u_int( XDR *xdrs, u_int * up );
```

Library

rpcLib.a

Description

xdr_u_int(), an XDR filter primitive, translates between a C **u_int** and its external representation.

xdrs points to the XDR stream handle; *up* points to the unsigned integer.

If successful, **xdr_u_int()** returns 1. Otherwise, **xdr_u_int()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_u_long()

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t xdr_u_long( XDR *xdrs, u_long *ulp );
```

Library

rpcLib.a

Description

xdr_u_long(), an XDR filter primitive, translates between a C **u_long** and its external representation.

xdrs points to the XDR stream handle; *ulp* points to the unsigned long integer.

If successful, **xdr_u_long()** returns 1. Otherwise, **xdr_u_long()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <types.h>
#include <rpc/rpc.h>

bool_t xdr_u_short( XDR *xdrs, u_short *usp );
```

Library

rpcLib.a

Description

xdr_u_short(), an XDR filter primitive, translates between a C **u_short** and its external representations.

xdrs points to the XDR stream handle; *usp* points to the unsigned short integer.

If successful, **xdr_u_short()** returns 1. Otherwise, **xdr_u_short()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_union()

Synopsis

```
#include <rpc/xdr.h>

bool_t xdr_union( XDR *xdrs, XDR *discmp, char *unp,
                  struct xdr_discrim *armchoices,
                  xdrproc_t (*defaultarm) );
```

Library

rpcLib.a

Description

xdr_union(), an XDR filter primitive, translates between a discriminated C union and its external representation.

xdr_union() first translates the discriminant of the union pointed to by *discmp*. The discriminant is always an **enum_t** value. Next, the union pointed to by *unp* is translated.

armchoices points to an array of **xdr_discrim** structures. Each structure contains an ordered pair of parameters (*value*, *proc*). If the discriminant of the union equals the associated *value*, then *proc* is called to translate the union. The end of the array of **xdr_discrim** structures is denoted by a routine of value NULL.

If the discriminant is not found the array of **xdr_discrim** structures, the procedure pointed to by *defaultarm* is called, if *defaultarm* is not NULL.

xdrs points to the XRD stream handle.

If successful, **xdr_union()** returns 1. Otherwise, **xdr_union()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <types.h>
#include <rpc/xdr.h>

bool_t xdr_vector( XDR *xdrs, char *arrp, u_int size, u_int elsize,
                  xdrproc_t elproc );
```

Library

rpcLib.a

Description

xdr_vector(), an XDR filter primitive, translates between a fixed-length array and its external representation.

xdrs points to the XDR stream handle; *arrp* points to the character array.

size specifies the element count of the array, and *elsize* specifies the element size.

elproc, an XDR filter, translates between the C form of the array elements and their external representation.

If successful, **xdr_vector()** returns 1. Otherwise, **xdr_vector()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdr_void()

Synopsis

```
#include <rpc/xdr.h>
bool_t xdr_void()
```

Library

rpcLib.a

Description

xdr_void(), which has no function parameters, may be passed to an RPC routines requiring a dummy parameter.

xdr_void() always returns 1.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/xdr.h>
bool_t xdr_wrapstring( XDR *xdrs, char **sp );
```

Library

rpcLib.a

Description

xdr_wrapstring(), an XDR primitive, calls:
 xdr_string(xdrs, sp, MAXUN.UNSIGNED).
MAXUN.UNSIGNED is the maximum value of an unsigned integer.
xdr_wrapstring() is useful because RPC passes a maximum of two XDR routines as parameters, and **xdr_string()** requires three.
xdrs points to the XDR stream handle; *sp* points to the string.
If successful, **xdr_wrapstring()** returns 1. Otherwise, **xdr_wrapstring()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **xdr_string()**, p. 989

xdrmem_create()

Synopsis

```
#include <types.h>
#include <rpc/xdr.h>

void xdrmem_create( XDR *xdrs, char *addr, u_int size, enum xdr_op op );
```

Library

rpcLib.a

Description

xdrmem_create() initializes, in local storage, the XDR stream pointed to by *xdrs*.

XDR stream data is written to or read from storage at the location pointed to by *addr*.

size specifies the size of the storage in bytes.

op specifies the XDR direction. The possible choices are **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <types.h>
#include <rpc/xdr.h>

void xdrrec_create( XDR *xdrs, u_int sendsize, u_int recvsz, char *handle,
                   int (*readit)(), (*writeit)() );
```

Library

rpcLib.a

Description

xdrrec_create() provides an XDR stream that can contain long sequences of records containing data in XDR form.

xdrrec_create() handles records in both the encoding and decoding directions.

xdrrec_create() initializes the XDR stream pointed to by *xdrs* parameter.

Note: The XDR stream implements an intermediate record stream; additional bytes in the stream provide record boundary information

sendsize and *recvsz* specify the sizes of the send and receive buffers. If either parameter is set to 0, **xdrrec_create()** assigns default buffer sizes.

handle points to the input/output buffer's handle, which is opaque.

readit points to the subroutine to call when a buffer needs to be filled. Similar to the read system call.

writeit points to the subroutine to call when a buffer needs to be flushed. Similar to the write system call.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xdrrec_endofrecord()

Synopsis

```
#include <rpc/xdr.h>

bool_t xdrrec_endofrecord( XDR *xdrs, bool_t sendnow );
```

Library

rpcLib.a

Description

xdrrec_endofrecord() marks outgoing data as a record.

xdrrec_endofrecord() can be invoked only on streams created using **xdrrec_create()**. Data in the output buffer is marked as a completed record; the output buffer is optionally written out if the value of *sendnow* is non-zero.

sendnow specifies whether the record should be flushed to the output TCP stream.

xdrs points to the XDR stream handle.

If successful, **xdrrec_endofrecord()** returns 1. Otherwise, **xdrrec_endofrecord()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **xdrrec_create()**, p. 999

Synopsis

```
#include <rpc/xdr.h>
bool_t xdrrec_eof( XDR *xdrs );
```

Library

rpcLib.a

Description

xdrrec_eof() checks the buffer for an input stream indicating end-of-file (EOF).

xdrrec_eof() can be invoked only on streams created using **xdrrec_create()**.

xdrs points to the XDR stream handle.

After **xdrrec_eof()** consumes the rest of the current record in the stream, **xdrrec_eof()** returns 1 if the stream has no more input; otherwise, **xdrrec_eof()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **xdrrec_create()**, p. 999

xdrrec_skiprecord()

Synopsis

```
#include <rpc/xdr.h>
bool_t xdrrec_skiprecord( XDR *xdrs );
```

Library

rpcLib.a

Description

xdrrec_skiprecord() moves the position of an input stream past the current record boundary and into the beginning of the next record of the stream.

xdrrec_skiprecord() can be invoked on only streams created using **xdrrec_create()**.

xdrrec_skiprecord() tells the XDR implementation that the remainder of the current record in the stream input buffer should be discarded.

xdrs points to the XDR stream handle.

If successful, **xdrrec_skiprecord()** returns 1. Otherwise, **xdrrec_skiprecord()** returns 0.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676
- **xdrrec_create()**, p. 999

Synopsis

```
#include <stdio.h>
#include <rpc/xdr.h>

void xdrstdio_create( XDR *xdrs, FILE *file, enum xdr_op op );
```

Library

rpcLib.a

Description

xdrstdio_create() initializes the XDR data stream pointed to by *xdrs*.

The XDR stream data is read from or written to the standard input or output pointed to by *file*.

op specifies the XDR direction. The possible choices are **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**; these constants are defined in the file **<rpc/xdr.h>**.

The destroy routine associated with the XDR stream calls **fflush()** on the file stream, but never calls **fclose()**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xprt_register()

Synopsis

```
#include <rpc/xdr.h>
void xprt_register( SVCXPRT *xprt );
```

Library

rpcLib.a

Description

xprt_register() registers the RPC service transport handle, pointed to by *xprt*, with the RPC program after the transport handle is created.

xprt_register() modifies the thread-private variable *svc_fds*.

Note: Service implementers do not usually need **xprt_register()**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	No
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

Synopsis

```
#include <rpc/xdr.h>
void xprt_unregister( SVCXPRT *xprt );
```

Library

rpcLib.a

Description

xprt_unregister() removes the RPC service transport handle pointed to by *xprt* from the RPC service program before the transport handle can be destroyed.

xprt_unregister() modifies the thread-private variable *svc_fds*.

Note: Service implementers do not usually need **xprt_unregister()**.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

- **rpc_thread_init()**, p. 676

xprt_unregister()

Index

Symbols

_callxlc() 88

A

abort() 50
abs() 51
accept() 52
access() 55
acos() 57
alarm() 58
align_h() 60
asctime() 62
asctime_r() 64
asin() 66
assert() 67
atan() 68
atan2() 69
atexit() 70
atof() 71
atoi() 72
atol() 74
auth_destroy(), AUTH_DESTROY() 76
authnone_create() 77
authunix_create() 78
authunix_create_default() 79

B

bind() 80
bootp_request() 82
bprintf() 84
bprintf_set() 85
bsearch() 86

C

calloc() 89
callrpc() 90
ceil() 91
cfgetispeed() 92
cfgetospeed() 93

cfsetispeed() 94
cfsetospeed() 95
chmod() 97
clearDisp() 98
clearerr() 99
cLib_init() 100
clnt_broadcast() 101
clnt_call(), CLNT_CALL() 103
clnt_control(), CLNT_CONTROL() 104
clnt_create() 106
clnt_destroy(), CLNT_DESTROY() 107
clnt_freeres(), CLNT_FREERES() 108
clnt_geterr(), CLNT_GETERR() 109
clnt_pcreateerror() 110
clnt_pereno() 111
clnt_perror() 112
clnt_spccreateerror() 113
clnt_spereno() 114
clnt_sperror() 115
clntraw_create() 116
clnttcp_create() 117
clntudp_bufcreate() 119
clntudp_create() 121
clock() 123
clock_getres() 124
clock_gettime() 125
clock_settime() 126
close() 127
closedir() 128
cond_dump() 129
cond_list() 130
connect() 132
cos() 134
cosh() 135
cpp_exit() 136
cpp_init() 137
cs_card_status() 138
cs_get_io_window() 139
cs_get_mem_window() 140
cs_init() 141
cs_int_install() 143
cs_int_query() 144

cs_pata_parm_init() 145
cs_set_ccr_offset() 147
cs_stateless_card_status() 148
ctime() 149
ctime_r() 150

D

dbbrkpt_clr() 151
dbbrkpt_cont() 152
dbbrkpt_disp() 153
dbbrkpt_find_inst() 154
dbbrkpt_init() 155
dbbrkpt_isbp() 156
dbbrkpt_next_inst() 157
dbbrkpt_set() 159
dbbrkpt_tclr() 160
dbbrkpt_tset() 161
dbbrkpt_write_inst() 162
dbmem_disp() 163
dbmem_listi() 164
dbstepi() 165
dbsymname_find() 166
dbwhere() 167
dcache_flush() 168
dcache_invalidate() 169
decompress() 170
dev_io_init() 171
device_install() 172
device_uninstall() 174
difftime() 175
div() 176
driver_install() 177
dtom() 178

E

enet_arpinut() 179
enet_arpresolve() 181
enet_attach() 183
errlog() 185
exit() 187
exp() 188

F

fabs() 189
fclose() 190
fcntl() 191
fdopen() 193
feof() 195
ferror() 196
fflush() 197
fgetc() 198
fgetpos() 199
fgets() 201
fileno() 202
flih_list() 203
flockfile() 205
floor() 208
fmod() 209
fopen() 210
format() 215
fpathconf() 217
fpreg_dump() 219
fprintf() 221
fputc() 222
fputs() 224
fread() 225
free() 228
freopen() 229
frexp() 230
fs_init() 231
fscanf() 234
fseek() 235
fsetpos() 237
fstat() 238
ftell() 239
ftp() 241
ftpd_start() 249
ftruncate() 250
ftrylockfile() 251
funlockfile() 252
fwrite() 253

G

get_myaddress() 254
getc() 255
getc_unlocked() 256
getchar() 257
getchar_unlocked() 258
getcwd() 261
getenv() 262
gethostbyaddr() 263
gethostbyname() 265
gethostlock() 266
gethostname() 267
gethostunlock() 268
getnetbyaddr() 270
getnetbyname() 272
getnetlock() 273
getnetunlock() 274
getopt() 275
getpeername() 277
getpid() 278
getprompt() 279
getprotobyname() 280
getprotobynumber() 281
getprotolock() 282
getprotounlock() 283
getrpcbyname() 285
getrpcbynumber() 286
getrpccent() 287
getrpcport() 288
gets() 289
getservbyname() 290
getservbyport() 291
getservlock() 293
getservunlock() 294
getsockname() 295
getsockopt() 296
gmtime() 299
gmtime_r() 301

H

heap_list() 302

I

if_attach() 303
ifconfig() 304
inbyte() 307
inet_addr() 308
inet_aton() 309
inet_lnaof() 311
inet_makeaddr() 312
inet_netof() 313
inet_network() 314
inet_ntoa() 316
inet_ntoa_r() 317
info_rtable() 318
inshort() 319
int_disable() 320
int_enable() 321
int_flihdisable() 322
int_flihenable() 324
int_flihregister() 325
int_getid() 326
int_install() 327
int_query() 329
inword() 331
ioctl() 332
isalnum() 333
isalpha() 335
isatty() 336
iscntrl() 337
isdigit() 338
isgraph() 339
islower() 340
isprint() 341
ispthreadid() 342
ispunct() 343
isspace() 344
issuspended() 345
istgid() 346
isupper() 347
isxdigit() 348

K

kda_dump() 349
kill() 350

L

labs() 351
ld() 352
ldexp() 354
ldiv() 355
ldr_ldinfo() 374
ldrDup_name() 356
ldrEntry_get() 359
ldrFree() 377
ldrIsexec() 360
ldrLib_init() 378
ldrLink() 361
ldrLoad() 363
ldrMoveData() 366
ldrMoveSym() 367
ldrMsg() 366, 367, 368
ldrQimage() 369
ldrQuery() 379
ldrResolve() 370
ldrStatus() 380
ldrSym_get() 371
ldrUnlink() 373
ldrUnload() 381
library_list() 382
link() 384
listen() 386
localeconv() 387
localtime() 388
localtime_r() 389
log() 390
log10() 391
longjmp() 392
lseek() 393

M

m_free() 394
m_freem() 395

M_PREPEND() 396

m_prepend() 397
malloc() 398
mblen() 399
mbstowcs() 401
mbtowc() 400
memchr() 402
memcmp() 403
memcpy() 404
memheap_alloc() 405
memheap_alloc_aligned() 406
memheap_alloc_pages() 407
memheap_extend() 408
memheap_free() 409
memheap_query() 410
memheap_realloc() 413
memheap_replace() 414
memmove() 415
mempool_alloc() 417
mempool_destroy() 418
mempool_free() 419
mempool_init() 420
mempool_query() 422
memset() 423
MFREE() 424
MGET() 425
MGETHDR() 426
mkdir() 427
mktime() 429
mlock() 430
mlockall() 431
mmap() 432
modf() 434
mq_close() 435
mq_dump() 436
mq_getattr() 437
mq_list() 438
mq_notify() 439
mq_open() 441
mq_receive() 446
mq_send() 449
mq_setattr() 452

mq_timedreceive() 453
mq_timedsend() 455
mq_unlink() 457
mtod() 458
munlock() 459
munlockall() 460
munmap() 461
mutex_dump() 462
mutex_list() 463
mutexattr_dump() 465

N

nanosleep() 466
net_init() 468
net_splimp() 469
net_splx() 470
nfs_authget() 471
nfs_authset() 472
nfs_dinit() 473
nfs_mount() 474
nfs_showmount() 475
nfs_umount() 476
np_compare_swap() 477

O

open() 478
opendir() 480
outbyte() 481
outshort() 482
outword() 483

P

package_install() 484
pata_init() 485
pathconf() 486
perror() 488
ping() 489
pmap_getmaps() 491
pmap_getport() 492
pmap_rmtcall() 493
pmap_set() 495
pmap_unset() 496

pool_list() 497
portmap_thread() 498
pow() 499
printf() 500
pthread_attr_destroy() 506
pthread_attr_getallocstack_np() 507
pthread_attr_getdetachstate() 508
pthread_attr_getfp_np() 509
pthread_attr_getinheritsched() 510
pthread_attr_getpr_np() 511
pthread_attr_getschedparam() 512
pthread_attr_getschedpolicy() 513
pthread_attr_getscope() 514
pthread_attr_getstackaddr() 515
pthread_attr_getstacksize() 516
pthread_attr_gettg_np() 517
pthread_attr_gettgid_np() 518
pthread_attr_init() 519
pthread_attr_setallocstack_np() 520
pthread_attr_setdetachstate() 521
pthread_attr_setfp_np() 522
pthread_attr_setinheritsched() 523
pthread_attr_setschedparam() 524
pthread_attr_setschedpolicy() 526
pthread_attr_setscope() 527
pthread_attr_setstackaddr() 528
pthread_attr_setstacksize() 529
pthread_attr_settg_np() 530
pthread_attr_settgid_np() 532
pthread_cancel() 533
pthread_cleanup_pop() 534
pthread_cleanup_push() 535
pthread_cond_broadcast() 538
pthread_cond_destroy() 539
pthread_cond_init() 540
pthread_cond_signal() 543
pthread_cond_timedwait() 544
pthread_cond_wait() 546
pthread_condattr_destroy() 548
pthread_condattr_getpshared() 549
pthread_condattr_init() 550
pthread_condattr_setpshared() 551

pthread_create() 552
pthread_detach() 554
pthread_equal() 555
pthread_errno_np() 556
pthread_exit() 557
pthread_getpr_np() 559
pthread_getschedparam() 560
pthread_getspecific() 561
pthread_getstackaddr_np() 562
pthread_join() 563
pthread_key_create() 564
pthread_key_delete() 567
pthread_kill() 568
pthread_mutex_destroy() 569
pthread_mutex_getprioceiling() 570
pthread_mutex_init() 571
pthread_mutex_lock() 572
pthread_mutex_setprioceiling() 573
pthread_mutex_timedlock() 574
pthread_mutex_trylock() 575
pthread_mutex_unlock() 576
pthread_mutexattr_destroy() 577
pthread_mutexattr_getprioceiling() 578
pthread_mutexattr_getprotocol() 579
pthread_mutexattr_getpshared() 580
pthread_mutexattr_init() 581
pthread_mutexattr_setprioceiling() 584
pthread_mutexattr_setprotocol() 585
pthread_mutexattr_setpshared() 586
pthread_once() 587
pthread_resume_np() 588
pthread_self() 589
pthread_setcancelstate() 590
pthread_setcanceltype() 591
pthread_setschedparam() 592
pthread_setspecific() 593
pthread_sigmask() 594
pthread_suspend_np() 596
pthread_testcancel() 597
pthread_tgcreate_np() 598
pthread_tgdestroy_np() 599
pthread_tggetkernel_np() 600

pthread_tggetspecific_np() 601
pthread_tgkeycreate_np() 603
pthread_tgkeydelete_np() 605
pthread_tgself_np() 606
pthread_tgsetspecific_np() 607
pthread_whattg_np() 608
putc() 609
putc_unlocked() 611
putchar() 612
putchar_unlocked() 613
puts() 615

Q

qsort() 616
queCreate() 617
queDeq() 619
queDeqNoCheck() 620
queEnq() 621
queEnqFront() 622
queInit() 623
queInsert() 624
queNewer() 625
queNewest() 626
queOlder() 627
queOldest() 628
queRemove() 629

R

raise() 630
rand() 631
rand_r() 632
read() 633
readdir() 636
readdir_r() 637
realloc() 638
recv() 640
recvfrom() 643
recvmsg() 645
reg_get() 647
reg_set() 648
registerrpc() 649
rename() 650

- rewind() 652
- rewinddir() 653
- rmdir() 654
- rms_delq() 655
- rms_init() 656
- rms_insq() 657
- rms_oldest() 658
- rms_start() 659
- rngBufGet() 662
- rngBufPut() 663
- rngCount() 664
- rngCreate() 665
- rngDelete() 666
- rngFlush() 667
- rngIsEmpty() 668
- rngIsFull() 669
- rootname_add() 670
- rootname_delete() 671
- rootname_get() 672
- rootname_query() 673
- route() 674
- rpc_thread_init() 676
- rsld_setschedparam() 677
- rsld_start() 678
- run() 679

S

- scanf() 681
- sched_getmask() 687
- sched_setmask() 688
- sched_yield() 689
- schednetisr() 690
- select() 691
- select_notify() 694
- select_redrive() 695
- sem_close() 696
- sem_destroy() 697
- sem_getvalue() 698
- sem_init() 699
- sem_open() 700
- sem_post() 705
- sem_timedwait() 706
- sem_trywait() 707
- sem_unlink() 708
- sem_wait() 709
- semaphore_dump() 710
- semaphore_list() 711
- send() 713
- sendmsg() 716
- sendto() 718
- setbuf() 720
- setenv_np() 721
- setjmp() 722
- setlocale() 723
- setprompt() 724
- setsockopt() 725
- setsysconf() 728
- setuname() 729
- setvbuf() 730
- shell() 732
- shell_threadid() 733
- shm_open() 734
- shm_unlink() 738
- shutdown() 739
- sigaddset() 740
- sigdelset() 741
- sigemptyset() 742
- sigfillset() 743
- sigismember() 744
- siglongjmp() 745
- signal() 746
- signal_list() 747
- sigpending() 748
- sigsetjmp() 750
- sigwait() 751
- sin() 753
- sinh() 754
- sleep() 755
- slip_attach() 756
- slip_resume() 758
- slip_suspend() 759
- socket() 760
- socket_services() 762
- sprintf() 763

- tftp_init() 873
- tg_dump() 874
- tg_dump_all() 876
- tg_list() 878
- thread_attr_dump() 879
- thread_debug() 880
- thread_dump() 881
- thread_info_list() 883
- thread_list() 884
- thread_name() 885
- time() 886
- time_request() 887
- timer_create() 888
- timer_delete() 890
- timer_dump() 891
- timer_getoverrun() 892
- timer_gettime() 893
- timer_list() 895
- timer_settime() 897
- timertick_notify() 899
- tmpfile() 900
- tmpnam() 901
- tn() 902
- tok_arpinput() 904
- tok_arpresolve() 905
- tolower() 906
- toupper() 907
- trace_checkall() 908
- trace_checklast() 909
- trace_copy() 910
- trace_get() 911
- trace_snapshot() 912
- trace_write() 913
- tty_init() 915
- tzset() 918

U

- uname() 920
- ungetc() 921
- unlink() 923
- utime() 924

V

- va_arg() 926
- va_end() 928
- va_start() 929
- vfprintf() 930
- vm_addrealpages() 932
- vm_allocate() 935
- vm_copy() 937
- vm_deallocate() 939
- vm_protect() 941
- vm_query() 943
- vm_read() 945
- vm_region() 947
- vm_remove_realpages() 949
- vm_translate() 951
- vm_write() 953
- vprintf() 955
- vsprintf() 956

W

- wcstombs() 959
- wctomb() 958
- write() 960

X

- xdr_accepted_reply() 962
- xdr_array() 963
- xdr_authunix_params() 964
- xdr_bool() 965
- xdr_bytes() 966
- xdr_callhdr() 967
- xdr_callmsg() 968
- xdr_char() 969
- xdr_destroy(), XDR_DESTROY() 970
- xdr_double() 971
- xdr_enum() 972
- xdr_float() 973
- xdr_free() 974
- xdr_getpos(), XDR_GETPOS() 975
- xdr_inline(), XDR_INLINE() 976
- xdr_int() 977

xdr_long() 978
xdr_opaque() 979
xdr_opaque_auth() 980
xdr_pmap() 981
xdr_pmaplist() 982
xdr_pointer() 983
xdr_reference() 984
xdr_rejected_reply() 985
xdr_replymsg() 986
xdr_setpos(), XDR_SETPOS() 987
xdr_short() 988
xdr_string() 989
xdr_u_char() 990
xdr_u_int() 991
xdr_u_long() 992
xdr_u_short() 993
xdr_union() 994
xdr_vector() 995
xdr_void() 996
xdr_wrapstring() 997
xdrmem_create() 998
xdrrec_create() 999
xdrrec_endofrecord() 1000
xdrrec_eof 1001
xdrrec_skiprecord() 1002
xdrstdio_create() 1003
xprt_register() 1004
xprt_unregister() 1005