

Transaction Processing Facility



# C/C++ Language Support User's Guide

*Version 4 Release 1*



Transaction Processing Facility



# C/C++ Language Support User's Guide

*Version 4 Release 1*

**Note!**

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xxi.

**Tenth Edition (June 2002)**

This is a major revision of, and obsoletes, SH31-0121-08 and all associated technical newsletters.

This edition applies to Version 4 Release 1 Modification Level 0 of IBM Transaction Processing Facility, program number 5748-T14, and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

IBM welcomes your comments. Address your comments to:

IBM Corporation  
TPF Systems Information Development  
Mail Station P923  
2455 South Road  
Poughkeepsie, NY 12601-5400  
USA

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1998, 2002. All rights reserved.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>Figures</b> . . . . .	xvii
<b>Tables</b> . . . . .	xix
<b>Notices</b> . . . . .	xxi
Trademarks . . . . .	xxi
<b>About This Book</b> . . . . .	xxiii
Who Should Read This Book . . . . .	xxiv
Conventions Used in the TPF Library . . . . .	xxiv
Related Information . . . . .	xxv
How to Send Your Comments . . . . .	xxvi
<b>TPF API Functions</b> . . . . .	1
Mapping Assembler Macros to C Functions . . . . .	3
abort—Terminate Program Abnormally . . . . .	7
access—Determine Whether a File Can Be Accessed . . . . .	9
addlc—Add a Block to the TPF Task Dispatch List . . . . .	11
alarm—Schedule an Alarm . . . . .	13
assert—Verify Condition or Print Diagnostic Message . . . . .	15
atexit—Call a Function at Normal Program Termination . . . . .	17
attac—Attach a Detached Working Storage Block . . . . .	19
attac_ext—Attach a Detached Working Storage Block . . . . .	21
attac_id—Attach a Detached Working Storage Block . . . . .	23
cebic_goto_bss—Change MDBF Subsystem to BSS . . . . .	25
cebic_goto_dbi—Change MDBF Subsystem to DBI . . . . .	26
cebic_goto_ssu—Change MDBF Subsystem . . . . .	27
cebic_restore—Restore Previously Saved DBI and SSU IDs . . . . .	28
cebic_save—Save the Current DBI and SSU IDs. . . . .	29
chdir—Change the Working Directory . . . . .	30
chmod—Change the Mode of a File or Directory . . . . .	32
chown—Change the Owner or Group of a File or Directory . . . . .	35
cifrc—Cipher Program Interface . . . . .	37
cincf—Control Program Interface. . . . .	38
cincf_fast—Fast Control Program Interface . . . . .	40
cincf_fast_ss—Fast Control Program Interface for Any Active Subsystem . . . . .	41
clearerr—Reset Error and End-of-File . . . . .	42
close—Close a File . . . . .	44
closedir—Close a Directory. . . . .	46
closelog—Close the System Control Log. . . . .	48
corhc—Define and Hold a Resource . . . . .	49
coruc—Unhold a Resource. . . . .	50
cratc—Search CRAS Status Table . . . . .	51
creat—Create a New File or Rewrite an Existing File . . . . .	54
credc, __CREDC—Create a Deferred Entry. . . . .	57
creec, __CREEC—Create a New ECB with an Attached Block. . . . .	59
cremc, __CREMC—Create an Immediate Entry . . . . .	62
cretc, __CRETc—Create a Time-Initiated Entry . . . . .	64
cretc_level, __CRETCL—Create a Time-Initiated Entry . . . . .	66
crexc, __CREXC—Create a Low-Priority Deferred Entry . . . . .	69
crosc_entrc—Cross-Subsystem to Enter a Program and then Return . . . . .	71
crusa—Free Core Storage Block If Held . . . . .	73
csonc—Convert System Ordinal Number. . . . .	75

dbzac–Attach TPF Application Requester Database Support Structure . . . . .	78
dbzdc–Detach TPF Application Requester Database Support Structure . . . . .	79
defrc–Defer Processing of Current Entry . . . . .	80
deleteCache–Delete a Logical Record Cache. . . . .	81
deleteCacheEntry–Delete a Cache Entry . . . . .	82
deqc–Dequeue from Resource . . . . .	84
detac–Detach a Working Storage Block from the ECB . . . . .	85
detac_ext–Detach a Working Storage Block from the ECB . . . . .	86
detac_id–Detach a Working Storage Block from the ECB . . . . .	88
dlayc–Delay Processing of Current Entry . . . . .	90
dllfree–Free the Supplied Dynamic Link Library (DLL) . . . . .	91
dllload–Load the DLL and Connect It to the Application . . . . .	93
dllqueryfn–Obtain a Pointer to a DLL Function . . . . .	94
dllqueryvar–Obtain a Pointer to a DLL Variable . . . . .	96
dup–Duplicate an Open File Descriptor . . . . .	97
dup2–Duplicate an Open File Descriptor to Another . . . . .	99
ecbptr–ECB Reference . . . . .	101
enqc–Define and Enqueue a Resource . . . . .	102
entdc, __ENTDC–Enter a Program and Drop Previous Programs . . . . .	104
entrc–Enter a Program with Expected Return . . . . .	106
evinc–Increment Count for Event. . . . .	108
evnqc–Query Event Status . . . . .	109
evntc–Define an Internal Event. . . . .	111
evnwc–Wait for Event Completion . . . . .	113
exit–Exit an ECB. . . . .	115
FACE, FACS–Low-Level File Address Compute Functions . . . . .	117
face_facs–File Address Generation . . . . .	120
fchmod–Change the Mode of a File or Directory by Descriptor . . . . .	123
fchown–Change the Owner and Group by File Descriptor. . . . .	125
fclose–Close File Stream. . . . .	127
fcntl–Control Open File Descriptors . . . . .	129
FD_CLR–Remove File Descriptor from the File Descriptor Set . . . . .	134
FD_COPY–Copy the File Descriptor Set . . . . .	135
FD_ISSET–Return a Value for the File Descriptor in the File Descriptor Set . . . . .	136
fdopen–Associate a Stream with an Open File Descriptor. . . . .	137
FD_SET–Add a File Descriptor to a File Descriptor Set . . . . .	139
FD_ZERO–Initialize the File Descriptor Set . . . . .	140
feof–Test End-of-File Indicator . . . . .	141
ferror–Test for Read/Write Errors . . . . .	143
fflush–Write Buffer to File . . . . .	144
fgetc–Read a Character . . . . .	146
fgetpos–Get File Position. . . . .	148
fgets–Read a String from a Stream . . . . .	150
filec–File a Record: Basic . . . . .	152
filec_ext–File a Record with Extended Options: Basic . . . . .	154
fileno–Get the File Descriptor from an Open Stream. . . . .	156
file_record–File a Record: Higher Level . . . . .	158
file_record_ext–File a Record with Extended Options: Higher Level . . . . .	161
filnc–File a Record with No Release . . . . .	165
filnc_ext–File a Record with No Release and Extended Options . . . . .	167
filuc–File and Unhold a Record . . . . .	170
filuc_ext–File and Unhold a Record with Extended Options . . . . .	172
findc–Find a Record . . . . .	174
findc_ext–Find a Record with Extended Options . . . . .	176
find_record–Find a Record . . . . .	178
find_record_ext–Find a Record with Extended Options . . . . .	181

finhc—Find and Hold a File Record . . . . .	185
finhc_ext—Find and Hold a File Record with Extended Options . . . . .	187
finwc—Find a File Record and Wait . . . . .	189
finwc_ext—Find a File Record and Wait with Extended Options . . . . .	191
fiwhc—Find and Hold a File Record and Wait . . . . .	193
fiwhc_ext—Find and Hold a File Record and Wait with Extended Options . . . . .	195
flipc—Interchange the Status of Two Data Levels . . . . .	197
flushCache—Flush the Cache Contents. . . . .	198
fopen—Open a File . . . . .	199
fprintf, printf, sprintf—Format and Write Data . . . . .	201
fputc—Write a Character . . . . .	209
fputs—Write a String. . . . .	211
fread—Read Items . . . . .	213
freopen—Redirect an Open File . . . . .	215
fscanf, scanf, sscanf—Read and Format Data . . . . .	217
fseek—Change File Position . . . . .	226
fsetpos—Set File Position . . . . .	228
fstat—Get Status Information about a File . . . . .	230
fsync—Write Changes to Direct Access Storage . . . . .	232
ftell—Get Current File Position . . . . .	234
ftok—Generate a Token . . . . .	235
ftruncate—Truncate a File. . . . .	237
fwrite—Write Items . . . . .	239
gdsnc—Get Data Set Entry . . . . .	241
gdsrc—Get General Data Set Record . . . . .	244
getc, getchar—Read a Character from Input Stream . . . . .	246
getcc—Obtain Working Storage Block . . . . .	248
getcwd—Get Path Name of the Working Directory . . . . .	252
getegid—Get the Effective Group ID . . . . .	254
getenv—Get Value of Environment Variables . . . . .	255
geteuid—Get the Effective User ID . . . . .	256
getfc—Obtain File Pool Address . . . . .	258
getgid—Get the Real Group ID . . . . .	260
getgrgid—Access the Group Database by ID. . . . .	261
getgrnam—Access the Group Database by Name . . . . .	263
getpc—Get Program and Lock in Core . . . . .	265
getpid—Obtain a Process ID. . . . .	267
getppid—Obtain the Parent Process ID. . . . .	268
getpwnam—Access the User Database by Name . . . . .	269
getpwuid—Access the User Database by User ID . . . . .	271
gets—Obtain Input String . . . . .	273
gettimeofday—Get Time . . . . .	275
getuid—Get the Real User ID . . . . .	276
glob—Address TPF Global Field or Record . . . . .	277
glob_keypoint—Keypoint TPF Global Field or Record . . . . .	279
glob_lock—Lock and Access Synchronizable TPF Global Field or Record . . . . .	280
glob_modify—Modify TPF Global Field or Record . . . . .	282
glob_sync—Synchronize TPF Global Field or Record. . . . .	284
glob_unlock—Unlock TPF Global Field or Record . . . . .	286
glob_update—Update TPF Global Field or Record. . . . .	288
global—Operate on TPF Global Field . . . . .	290
gysc—Get Storage from the System Heap . . . . .	293
inqrc—Convert Resource Application Interface . . . . .	295
keyrc—Restore Protection Key . . . . .	296
keyrc_okey—Restore Original Protection Key . . . . .	297
kill—Send a Signal to a Process . . . . .	298

levtest—Test Core Level for Occupied Condition . . . . .	300
link—Create a Link to a File . . . . .	302
lockc—Lock a Resource . . . . .	305
lodc—Check System Load and Mark ECB . . . . .	306
lodc_ext—Check System Load and Mark ECB with Extended Options . . . . .	308
longc—Set Entry Maximum Existence Time . . . . .	312
longjmp—Restore Stack Environment . . . . .	313
lseek—Change the Offset of a File . . . . .	315
lstat—Get Status of a File or Symbolic Link . . . . .	317
mail—Process Internet Mail . . . . .	320
maskc—Modify Program Status Word (PSW) Mask Bits. . . . .	325
mkdir—Make a Directory . . . . .	326
mkfifo—Make a FIFO Special File. . . . .	328
mknod—Make a Character Special File. . . . .	331
MQBACK—Back Out a Queue . . . . .	333
MQCLOSE—Close a Queue. . . . .	335
MQCMIT—Commit a Queue . . . . .	337
MQCONN—Connect Queue Manager . . . . .	339
MQDISC—Disconnect Queue Manager. . . . .	341
MQGET—Get Message from an Open Queue . . . . .	343
MQINQ—Inquire about Object Attributes . . . . .	349
MQOPEN—Open a Queue . . . . .	355
MQPUT—Put a Message on an Open Queue . . . . .	359
MQPUT1—Put a Single Message on a Queue . . . . .	365
MQSET—Set Object Attributes . . . . .	372
newCache—Create a New Logical Record Cache . . . . .	376
numbc—Query Number of Storage Blocks Available . . . . .	379
open—Open a File . . . . .	380
opendir—Open a Directory . . . . .	384
openlog—Open the System Control Log . . . . .	386
pausc—Control System Multiprocessor Environment . . . . .	387
pause—Wait for a Signal . . . . .	388
perror—Write Error Message to Standard Error Stream . . . . .	389
pipe—Create an Unnamed Pipe . . . . .	391
postc—Mark Event Element Completion . . . . .	395
printf—Format and Write Data . . . . .	397
progc—Return Program Allocation Table (PAT) Slot Address . . . . .	398
putc, putchar—Write a Character . . . . .	400
puts—Put String to Standard Output Stream . . . . .	402
raisa—General File Get File Address. . . . .	404
raise—Raise Condition . . . . .	406
rcunc—Release Core Block and Unhold File Record . . . . .	408
read—Read from a File . . . . .	410
readCacheEntry—Read a Cache Entry . . . . .	413
readdir—Read an Entry from a Directory . . . . .	415
readlink—Read the Value of a Symbolic Link. . . . .	417
rehka—Rehook Core Block . . . . .	419
relcc—Release Working Storage Block . . . . .	421
relfc—Release File Pool Storage . . . . .	423
relpc—Release Program from Core Lock . . . . .	425
remove—Delete a File . . . . .	427
rename—Rename a File . . . . .	428
rewind—Set File Position to Beginning of File . . . . .	431
rewinddir—Reposition a Directory Stream to the Beginning . . . . .	433
ridcc—SNA RID Conversions . . . . .	435
rlcha—Release Chained File Records . . . . .	437



rmdir—Remove a Directory . . . . .	439
route—Route a Message . . . . .	441
rsysc—Release Storage from the System Heap. . . . .	443
rvtcc—Search Resource Vector Table (RVT) Entries . . . . .	445
scanf—Scan Input for Variables . . . . .	447
selec—Select a Thread Application Interface . . . . .	448
select—Monitor Read, Write, and Exception Status . . . . .	450
serrc_op—Issue System Error: Operational . . . . .	454
serrc_op_ext—Issue System Error Extended: Operational . . . . .	456
serrc_op_slst—Issue System Error SLIST: Operational . . . . .	458
setbuf—Control Buffering . . . . .	460
setgid—Set the Effective Group ID . . . . .	462
setenv—Add, Change, or Delete an Environment Variable . . . . .	463
seteuid—Set the Effective User ID . . . . .	466
setgid—Set the Group ID to a Specified Value . . . . .	468
setjmp—Preserve Stack Environment . . . . .	470
setuid—Set the Real User ID . . . . .	472
setvbuf—Control Buffering . . . . .	474
shmat—Attach Shared Memory. . . . .	476
shmctl—Shared Memory Control . . . . .	478
shmdt—Detach Shared Memory . . . . .	481
shmget—Allocate Shared Memory. . . . .	483
sigaction—Examine and Change Signal Action . . . . .	486
sigaddset—Add a Signal to a Signal Set . . . . .	490
sigdelset—Delete a Signal from a Signal Set. . . . .	491
sigemptyset—Initialize and Empty a Signal Set . . . . .	492
sigfillset—Initialize and Fill a Signal Set. . . . .	493
sigismember—See If a Signal Is a Member of a Signal Set . . . . .	494
signal—Install Signal Handler . . . . .	495
sigpending—Examine Pending Signals . . . . .	498
sigprocmask—Examine and Change Blocked Signals . . . . .	499
sigsuspend—Set Signal Mask and Wait for a Signal . . . . .	502
sipcc—System Interprocessor Communication . . . . .	504
sleep—Suspend the Calling Process. . . . .	507
snpc—Issue Snapshot Dump . . . . .	508
sonic—Obtain Symbolic File Address Information . . . . .	511
stat—Get File Information . . . . .	513
strerror—Get Pointer to Run-Time Error Message . . . . .	517
swisc_create—Create New ECB on Specified I-Stream . . . . .	518
symlink—Create a Symbolic Link to a Path Name . . . . .	521
syslog—Send a Message to the Control Log . . . . .	523
systc—Test System Generation Options . . . . .	526
system—Execute a Command . . . . .	527
tancc—Transaction Anchor Table Control . . . . .	530
tape_access—Access a Tape . . . . .	532
tape_close—Close a General Tape . . . . .	534
tape_cntl—Tape Control . . . . .	535
tape_open—Open a General Tape . . . . .	537
tape_read—Read a Record From General Tape . . . . .	538
tape_write—Write a Record to General Tape . . . . .	540
tasnc—Assign General Tape to ECB. . . . .	541
tbspc—Backspace General Tape and Wait . . . . .	542
tclsc—Close a General Tape. . . . .	544
tdspc—Display Tape Status . . . . .	545
tdspc_q—Display Tape Queue Length . . . . .	546
tdspc_v—Retrieve VOLSER for a Specified Tape Name. . . . .	547

tdtac—Issue a User-Specified Data Transfer CCW. . . . .	549
tmpfile—Create a Temporary File . . . . .	551
tmpnam—Produce a Temporary Name . . . . .	552
tmslc—ECB Time Slice Facility . . . . .	553
topnc—Open a General Tape . . . . .	556
tourc—Write Real-Time Tape Record/Release Storage Block . . . . .	557
toutc—Write Real-Time Tape Record. . . . .	558
tpcnc—Issue a User-Specified Control Operation CCW . . . . .	560
TPF_CALL_BY_NAME, TPF_CALL_BY_NAME_STUB—Call (Enter) a Program by Name . . . . .	562
tpf_cfconc—Connect to a Coupling Facility List or Cache Structure . . . . .	564
tpf_cfdisc—Disconnect from a Coupling Facility List or Cache Structure . . . . .	570
tpf_cresc—Create Synchronous Child ECBs . . . . .	572
tpf_decb_create—Create a Data Event Control Block. . . . .	576
tpf_decb_locate—Locate a Data Event Control Block. . . . .	578
tpf_decb_release—Release a Data Event Control Block. . . . .	580
tpf_decb_swapblk—Swap a Storage Block with a Data Event Control Block . . . . .	582
tpf_decb_validate—Validate a Data Event Control Block. . . . .	584
tpf_dlckc — Modify Lock and Input/Output Interrupt Status. . . . .	585
tpf_esfac—Obtain Extended Symbolic File Address Information . . . . .	586
tpf_fac8c—Low-Level File Address Compute Functions . . . . .	587
tpf_faczc—File Address Calculation . . . . .	589
tpf_fa4x4c—Convert a File Address . . . . .	592
tpf_fork—Create a Child Process . . . . .	594
tpf_genlc—Generate a Data List . . . . .	599
tpf_gsvac—Convert an EVM Address to an SVM Address . . . . .	602
tpf_help—Issue Help Messages . . . . .	603
tpf_is_RPCServer_auto_restarted—Query If an RPC Server Is Restarted . . . . .	607
tpf_itrpc—Send Simple Network Management Protocol User Trap . . . . .	608
tpf_lemic —Lock Entry Management Interface . . . . .	612
tpf_movec—Move Data between EVM and SVM . . . . .	616
tpf_movec_EVM—Move Data from One EVM to Another EVM . . . . .	618
tpf_msg—Issue a Message . . . . .	619
tpf_process_signals—Process Outstanding Signals . . . . .	622
tpf_rcrfc—Release a Core Block and File Address. . . . .	624
tpf_RPC_options—Set TPF- Specific Thread Options. . . . .	626
tpf_sawnc—Wait for Event Completion with Signal Awareness . . . . .	627
tpf_select_bsd—Indicates Read, Write, and Exception Status. . . . .	629
tpf_snmp_BER_encode—Encode SNMP Variables in BER Format. . . . .	632
tpf_STCK—Store Clock . . . . .	634
tpf_tcpip_message_cnt—Update the Message Counters for TCP/IP Applications . . . . .	635
tpf_tm_getToken—Get the Unique Token for the Current Transaction . . . . .	636
tpf_vipac—Move a VIPA to Another Processor . . . . .	637
tpf_yieldc—Yield Control . . . . .	639
tprdc—Read General Tape Record . . . . .	641
trewc—Rewind General Tape and Wait . . . . .	643
trsvc—Reserve General Tape . . . . .	644
tsync—Synchronize a Tape . . . . .	645
twrtc—Write a Record to General Tape . . . . .	646
tx_begin—Begin a Global Transaction . . . . .	647
tx_commit—Commit a Global Transaction . . . . .	649
tx_open—Open a Set of Resource Managers . . . . .	651
tx_resume_tpf—Resume a Global Transaction . . . . .	652
tx_rollback—Roll Back a Global Transaction . . . . .	654
tx_suspend_tpf—Suspend a Global Transaction. . . . .	655
uatbc—MDBF User Attribute Reference Request . . . . .	657

umask–Set the File Mode Creation Mask . . . . .	659
unfrcl–Unhold a File Record . . . . .	661
unfrcl_ext–Unhold a File Record with Extended Options . . . . .	662
ungetc–Push Character to Input Stream . . . . .	664
unhkl–Unhook Core Block . . . . .	666
unlink–Remove a Directory Entry . . . . .	668
unlck–Unlock a Resource . . . . .	670
unsetenv–Delete an Environment Variable . . . . .	671
updateCacheEntry–Add a New or Update an Existing Cache Entry . . . . .	673
utime–Set File Access and Modification Times . . . . .	676
vfprintf–Format and Print Data to a Stream . . . . .	678
vprintf–Format and Print Data to stdout . . . . .	680
vsprintf–Format and Print Data to a Buffer . . . . .	682
wait–Wait for Status Information from a Child Process . . . . .	684
waitc–Wait For Outstanding I/O Completion . . . . .	686
waitpid–Obtain Status Information from a Child Process . . . . .	687
WEXITSTATUS–Obtain Exit Status of a Child Process . . . . .	690
wgtac–Locate Terminal Entry . . . . .	691
wgtac_ext–Locate Terminal Entry with Extended Options . . . . .	692
WIFEXITED–Query Status to See If a Child Process Ended Normally . . . . .	694
WIFSIGNALED–Query Status to See If a Child Process Ended Abnormally . . . . .	695
write–Write Data to a File Descriptor . . . . .	696
WTERMSIG–Determine Which Signal Caused the Child Process to Exit . . . . .	699
wtopc–Send System Message . . . . .	700
wtopc_insert_header–Save Header for wtopc . . . . .	704
wtopc_routing_list–Save Routing List for wtopc . . . . .	706
wtopc_text–Send System Message . . . . .	707
xa_commit–Commit Work Done for a Transaction Branch . . . . .	708
xa_end–End Work Performed for a Transaction Branch . . . . .	710
xa_open–Open a Resource Manager . . . . .	712
xa_prepare–Prepare to Commit . . . . .	714
xa_recover–Get a List of Prepared Transaction Branches . . . . .	716
xa_rollback–Roll Back Work Done for a Transaction Branch . . . . .	718
xa_start–Start Work for a Transaction Branch . . . . .	720
<b>TPF/APPC Basic Conversation Functions . . . . .</b>	<b>723</b>
General Format . . . . .	724
Return Codes for Basic Conversation Functions . . . . .	731
Programming Considerations for Basic Conversation Functions . . . . .	735
tppc_activate_on_confirmation–Activate a Program after Confirmation Received . . . . .	737
tppc_activate_on_receipt–Activate a Program after Information Received . . . . .	741
tppc_allocate–Allocate a Conversation . . . . .	745
tppc_confirm–Send a Confirmation Request . . . . .	750
tppc_confirmed–Send a Confirmation Reply . . . . .	753
tppc_deallocate–Deallocate a Conversation . . . . .	755
tppc_flush–Flush Data from Local LU Buffer . . . . .	759
tppc_get_attributes–Get Information about a Conversation . . . . .	761
tppc_get_type–Get Conversation Type . . . . .	763
tppc_post_on_receipt–Set Posting Active for a Conversation . . . . .	765
tppc_prepare_to_receive–Change to Receive State . . . . .	768
tppc_receive–Receive Information . . . . .	771
tppc_request_to_send–Request Change to Send State . . . . .	777
tppc_send_data–Send Data to Remote Transaction Program . . . . .	779
tppc_send_error–Send Error Notification . . . . .	782
tppc_test–Test Conversation . . . . .	786
tppc_wait–Wait for Posting . . . . .	789

<b>TPF/APPC Mapped Conversation Functions</b>	793
TPF/APPC Mapped Conversation Interface Overview	793
Return Codes for Mapped Conversation Functions	796
cmaccp—Accept a Conversation	798
cmalloc—Allocate a Conversation	800
cmcfm—Send a Confirmation Request	803
cmcfmd—Send a Confirmation Reply	805
cmdeal—Deallocate a Conversation	807
cmecs—Extract the Conversation State	810
cmemn—Extract the Mode Name	812
cmepln—Extract the Partner LU Name	814
cmesl—Extract the Sync Level	816
cmflus—Flush Data from Buffer of Local LU	818
cminit—Initialize a Conversation	820
cmptr—Prepare to Receive Data	823
cmrcv—Receive Information	825
cmrts—Request Change to Send State	829
cmsdt—Set the Deallocate_Type Characteristic	831
cmsed—Set the Error_Direction Characteristic	833
cmsend—Send Data to Remote Transaction Program	835
cmserr—Send Error Notification	838
cmsmn—Set the Mode_Name Characteristic	842
cmspln—Set the Partner_LU_Name Characteristic	844
cmsptr—Set the Prepare_To_Receive_Type Characteristic	846
cmsrc—Set the Return_Control Characteristic	848
cmssl—Set the Sync_Level Characteristic	850
cmsst—Set the Send_Type Characteristic	852
cmstpn—Set the TP_Name Characteristic	854
cmtrts—Test for Request_To_Send Notification	856
 <b>TPF Collection Support: Environment Functions</b>	 859
Type Definitions	859
Retrieving Error Information	860
Error Code Summary	861
TO2_createEnv—Create an Environment Block	874
TO2_deleteEnv—Delete an Environment Block	876
TO2_getErrorCode—Retrieve the Error Code Value	877
TO2_getErrorText—Retrieve the Associated Error Text	879
 <b>TPF Collection Support: Non-Cursor</b>	 881
Collection Support for Non-Cursor APIs	881
TO2_add—Add an Element to a Collection	883
TO2_addAllFrom—Add All from Source Collection	886
TO2_addAtIndex—Insert an Element in an Sequence Collection	888
TO2_addKeyPath—Add a Key Path to a Collection	890
TO2_addRecoupIndexEntry—Add an Entry to a Recoup Index	893
TO2_asOrderedCollection—Return Contents As Ordered	896
TO2_asSequenceCollection—Return Contents As Sequence	897
TO2_associateRecoupIndexWithPID—Create a PID to Index Association	899
TO2_asSortedCollection—Return Contents As Sorted Bag	901
TO2_at—Return the Specified Element by Index	903
TO2_atKey—Return the Specified Element by Key	905
TO2_atKeyPut—Update the Specified Element Using a Key	907
TO2_atKeyWithBuffer—Return the Specified Element	909
TO2_atNewKeyPut—Add a New Key and Element	911
TO2_atPut—Update the Specified Element	913

TO2_atRBA–Retrieve Data from a BLOB . . . . .	915
TO2_atRBAPut–Store Data in a BLOB. . . . .	917
TO2_atRBAWithBuffer–Retrieve Data from a BLOB . . . . .	920
TO2_atWithBuffer–Retrieve an Element from a Collection. . . . .	922
TO2_capture–Capture a Collection to an External Device. . . . .	924
TO2_copyCollection–Make a Persistent Copy of the Collection. . . . .	927
TO2_copyCollectionTemp–Make a Temporary Copy of the Collection . . . . .	929
TO2_copyCollectionWithOptions–Make a Copy Using Options . . . . .	931
TO2_createArray–Create a Persistent Array Collection . . . . .	933
TO2_createArrayTemp–Create a Temporary Array Collection . . . . .	935
TO2_createArrayWithOptions–Create an Empty Array Collection . . . . .	937
TO2_createBag–Create a Persistent Bag Collection. . . . .	939
TO2_createBagTemp–Create a Temporary Bag Collection . . . . .	941
TO2_createBagWithOptions–Create an Empty Bag Collection . . . . .	943
TO2_createBLOB–Create a Persistent BLOB Collection . . . . .	945
TO2_createBLOBTemp–Create a Temporary BLOB Collection . . . . .	947
TO2_createBLOBWithOptions–Create an Empty BLOB Collection. . . . .	949
TO2_createDictionary–Create a Persistent Dictionary Collection . . . . .	951
TO2_createDictionaryTemp–Create a Temporary Dictionary . . . . .	952
TO2_createDictionaryWithOptions–Create an Empty Dictionary Collection . . . . .	953
TO2_createKeyBag–Create a Persistent Key Bag Collection. . . . .	954
TO2_createKeyBagTemp–Create a Temporary Key Bag Collection . . . . .	956
TO2_createKeyBagWithOptions–Create an Empty Key Bag Collection with Options . . . . .	958
TO2_createKeyedLog–Create a Persistent Keyed Log Collection . . . . .	960
TO2_createKeyedLogTemp–Create a Temporary Keyed Log Collection. . . . .	962
TO2_createKeyedLogWithOptions–Create an Empty Keyed Log Collection with Options . . . . .	964
TO2_createKeySet–Create a Persistent Key Set Collection . . . . .	966
TO2_createKeySetTemp–Create a Temporary Key Set Collection . . . . .	968
TO2_createKeySetWithOptions–Create an Empty Key Set Collection . . . . .	970
TO2_createKeySortedBag–Create a Persistent Key Sorted Bag Collection . . . . .	972
TO2_createKeySortedBagTemp–Create a Temporary Sorted Key Bag . . . . .	974
TO2_createKeySortedBagWithOptions–Create an Empty Key Sorted Bag with Options . . . . .	976
TO2_createKeySortedSet–Create an Empty Persistent Key Sorted Set Collection . . . . .	978
TO2_createKeySortedSetTemp–Create a Temporary Key Sorted Set . . . . .	980
TO2_createKeySortedSetWithOptions–Create an Empty Key Sorted Set Collection . . . . .	982
TO2_createLog–Create an Empty Persistent Log Collection . . . . .	984
TO2_createLogTemp–Create a Temporary Log Collection. . . . .	986
TO2_createLogWithOptions–Create an Empty Log Collection with Options . . . . .	988
TO2_createOptionList–Create a TPF Collection Support Option List . . . . .	990
TO2_createOrder–Create a Persistent Ordered Collection . . . . .	995
TO2_createOrderTemp–Create a Temporary Ordered Collection . . . . .	996
TO2_createOrderWithOptions–Create an Empty Ordered Collection with Options . . . . .	997
TO2_createRecoupIndex–Create a Recoup Index . . . . .	998
TO2_createSequence–Create a Persistent Sequence Collection . . . . .	1000
TO2_createSequenceTemp–Create a Temporary Sequence Collection . . . . .	1002
TO2_createSequenceWithOptions–Create an Empty Sequence Collection with Options . . . . .	1004
TO2_createSet–Create a Persistent Set Collection. . . . .	1006
TO2_createSetTemp–Create a Temporary Set for the Collection . . . . .	1008
TO2_createSetWithOptions–Create an Empty Set Collection with Options . . . . .	1010



TO2_createSort—Create a Persistent Sorted Collection . . . . .	1012
TO2_createSortedBag—Create an Empty Persistent Sorted Bag Collection . . . . .	1013
TO2_createSortedBagTemp—Create a Temporary Sorted Bag Collection . . . . .	1015
TO2_createSortedBagWithOptions—Create an Empty Sorted Bag Collection with Options . . . . .	1017
TO2_createSortedSet—Create a Persistent Sorted Set Collection . . . . .	1019
TO2_createSortedSetTemp—Create a Temporary Sorted Set Collection . . . . .	1021
TO2_createSortedSetWithOptions—Create an Empty Sorted Set Collection with Options . . . . .	1023
TO2_createSortTemp—Create a Temporary Sorted Collection . . . . .	1026
TO2_createSortWithOptions—Create an Empty Sorted Collection with Options . . . . .	1027
TO2_definePropertyForPID—Define or Change a Property for a Collection . . . . .	1028
TO2_definePropertyWithModeForPID—Define a Property with a Mode . . . . .	1031
TO2_deleteAllPropertiesFromPID—Delete All Defined Properties . . . . .	1034
TO2_deleteCollection—Delete a Collection . . . . .	1036
TO2_deletePropertyFromPID—Delete a Property. . . . .	1038
TO2_deleteRecoupIndex—Delete a Recoup Index . . . . .	1040
TO2_deleteRecoupIndexEntry—Delete an Entry from an Index. . . . .	1041
TO2_getAllPropertyNamesFromPID—Return Property Names . . . . .	1043
TO2_getBLOB—Retrieve the Contents of a BLOB . . . . .	1045
TO2_getBLOBwithBuffer—Retrieve Contents of a BLOB Using a Passed Buffer . . . . .	1047
TO2_getCollectionAccessMode — Retrieve the Access Mode of a Collection . . . . .	1049
TO2_getCollectionKeys—Get the Primary Key Values of the Collection. . . . .	1051
TO2_getCollectionType—Get the Type Value of the Collection . . . . .	1053
TO2_getDRprotect — Retrieve Dirty-Reader Protection Status . . . . .	1055
TO2_getMaxDataLength—Retrieve the Maximum Data Length Value . . . . .	1057
TO2_getMaxKeyLength—Retrieve the Maximum Key Length Value . . . . .	1058
TO2_getPropertyValueFromPID—Retrieve a Property Value. . . . .	1060
TO2_getSortFieldValues—Retrieve the Sort Field Values . . . . .	1062
TO2_includes—Search Collection for Element Equal to the Argument . . . . .	1064
TO2_isCollection—Test If PID Is for a Collection . . . . .	1066
TO2_isPropertyDefinedForPID—Test If a Property Is Already Defined . . . . .	1068
TO2_isTemp—Determine If Collection Is Temporary or Persistent. . . . .	1070
TO2_makeEmpty—Delete All Elements from a Collection. . . . .	1072
TO2_maxEntry—Return the Maximum Number of Elements in a Collection . . . . .	1074
TO2_readOnly—Get the Read-Only Attribute of the Collection . . . . .	1076
TO2_removeIndex—Remove Element from the Index . . . . .	1078
TO2_removeKey—Remove the Key-Entry Pair . . . . .	1080
TO2_removeKeyPath—Remove a Key Path from a Collection . . . . .	1082
TO2_removeRBA—Remove an Area from a BLOB . . . . .	1084
TO2_removeRecoupIndexFromPID—Remove a PID to Index Association . . . . .	1086
TO2_removeValue—Remove Value. . . . .	1088
TO2_removeValueAll—Remove All Matching Values from the Collection . . . . .	1090
TO2_replaceBLOB—Replace the Contents of a BLOB with New Data . . . . .	1092
TO2_restore—Restore a Previously Captured Collection . . . . .	1094
TO2_restoreAsTemp—Restore a Collection as a Temporary Collection . . . . .	1096
TO2_restoreWithOptions—Restore a Collection Using the Specified Options . . . . .	1098
TO2_setCollectionAccessMode — Set the Access Mode of a Collection . . . . .	1101
TO2_setDRprotect — Set Dirty-Reader Protection On. . . . .	1103
TO2_setReadOnly—Set the Read-Only Attribute of the Collection. . . . .	1105
TO2_setSize—Set the Size of the BLOB . . . . .	1107
TO2_size—Return the Size of the Collection . . . . .	1109
TO2_writeNewBLOB—Create a New BLOB and Add the Passed Data . . . . .	1111
<b>TPF Collection Support: Cursors . . . . .</b>	<b>1113</b>

Supported Collection Classes for Cursor APIs. . . . .	1113
TO2_addAtCursor-Insert an Element in a Sequence Collection . . . . .	1115
TO2_allElementsDo-Iterate over All Elements. . . . .	1117
TO2_atCursor-Return the Element Pointed to by the Cursor . . . . .	1119
TO2_atCursorPut-Store Element Where Cursor Points . . . . .	1121
TO2_atCursorWithBuffer-Return the Element Pointed to by the Cursor . . . . .	1123
TO2_atEnd-Test If the Cursor Is at the End of the Collection . . . . .	1125
TO2_atLast-Test If the Cursor Points to the Last Element . . . . .	1127
TO2_createCursor-Create a Nonlocking Cursor . . . . .	1129
TO2_createReadWriteCursor-Create a Locking Cursor . . . . .	1131
TO2_cursorMinus-Decrement Cursor to Previous Element . . . . .	1133
TO2_cursorPlus-Increment Cursor to Next Element . . . . .	1135
TO2_deleteCursor-Delete a Previously Created Cursor . . . . .	1137
TO2_first-Point Cursor to First Element . . . . .	1139
TO2_getCurrentKey-Retrieve the Current Key . . . . .	1141
TO2_getCurrentKeyWithBuffer-Retrieve the Current Key in the Buffer. . . . .	1143
TO2_index-Return Current Position Index Value. . . . .	1145
TO2_isEmpty-Test If Collection Is Empty . . . . .	1147
TO2_key-Return Current Key Value . . . . .	1149
TO2_keyWithBuffer-Return Current Key Value in the Buffer . . . . .	1151
TO2_last-Point Cursor at Last Element . . . . .	1153
TO2_locate-Locate Key and Point Cursor to Its Element . . . . .	1155
TO2_more-Test for More Elements to Process . . . . .	1157
TO2_next-Increment and Return the Next Element. . . . .	1159
TO2_nextPut-Store This Element As the Next Element . . . . .	1161
TO2_nextRBAfor-Return the Next Specified Number of Bytes. . . . .	1163
TO2_nextWithBuffer-Increment and Return the Next Element. . . . .	1165
TO2_peek-Return the Next Element with No Cursor Movement . . . . .	1168
TO2_peekWithBuffer-Return the Next Element . . . . .	1170
TO2_previous-Return the Previous Element . . . . .	1172
TO2_previousWithBuffer-Return the Previous Element in the Specified Buffer . . . . .	1174
TO2_remove-Remove the Element Pointed to by the Cursor . . . . .	1176
TO2_reset-Reset Cursor to Point to First Element . . . . .	1178
TO2_setKeyPath-Set a Cursor to Use a Specific Key Path. . . . .	1180
TO2_setPositionIndex-Point Cursor at Specified Position . . . . .	1182
TO2_setPositionValue-Point Cursor at Specified Element . . . . .	1184
 <b>TPF Collection Support: Dictionary.</b> . . . . .	 1187
Data Store Application Dictionary . . . . .	1187
TO2_atDSdictKey-Retrieve the Element Using the Specified Key . . . . .	1188
TO2_atDSdictKeyPut-Replace the Element Using the Key . . . . .	1190
TO2_atDSdictNewKeyPut-Store the Element Using the New Key . . . . .	1192
TO2_atDSsystemKey-Retrieve the Element Using the Specified Key . . . . .	1194
TO2_atDSsystemKeyPut-Replace the Element Using the Key . . . . .	1196
TO2_atDSsystemNewKeyPut-Store the Element Using the New Key . . . . .	1198
TO2_atTPFKey-Retrieve the Element Using the Specified Key . . . . .	1200
TO2_atTPFKeyPut-Replace the Element Using the Specified Key . . . . .	1202
TO2_atTPFNewKeyPut-Store the Element Using the New Key . . . . .	1204
TO2_atTPFsystemKey-Retrieve the Element Using the Specified Key . . . . .	1206
TO2_atTPFsystemKeyPut-Replace the Element Using the Specified Key . . . . .	1208
TO2_atTPFsystemNewKeyPut-Store the Element Using the New Key . . . . .	1210
TO2_getDSdictPID-Get the Dictionary PID of a Data Store . . . . .	1212
TO2_getTPFDictPID-Get the PID of the TPF Dictionary . . . . .	1214
TO2_removeDSdictKey-Remove the Element Using the Key . . . . .	1215
TO2_removeDSsystemKey-Remove the Element Using the Key . . . . .	1217
TO2_removeTPFKey-Remove the Element from the TPF Dictionary . . . . .	1219

TO2_removeTPFsystemKey–Remove the Element from the TPF System Dictionary . . . . .	1221
<b>TPF Collection Support: Browser . . . . .</b>	<b>1223</b>
TO2_atBrowseKey–Retrieve the Element Using the Specified Key from the Browser Dictionary. . . . .	1224
TO2_atBrowseKeyPut–Replace the Element Using the Specified Key from the Browser Dictionary. . . . .	1226
TO2_atBrowseNewKeyPut–Store the Element Using the Specified Key in the Browser Dictionary. . . . .	1228
TO2_changeDD–Change a Data Definition. . . . .	1230
TO2_changeDS–Change the Attributes of a Data Store . . . . .	1232
TO2_convertClassName–Convert EBCDIC Class Name to Class Index . . . . .	1234
TO2_convertMethodName–Convert Method Name to Entry Address . . . . .	1236
TO2_createDD–Create a Data Definition . . . . .	1238
TO2_createDS–Create a Data Store . . . . .	1240
TO2_createDSwithOptions–Create a Data Store with an Option List . . . . .	1242
TO2_createPIDinventoryKey–Create PID Inventory Key from PID . . . . .	1244
TO2_defineBrowseNameForPID–Assign a Browse Name to a Collection . . . . .	1246
TO2_deleteBrowseName–Delete a Collection Name from the Browser Dictionary . . . . .	1248
TO2_deleteDD–Delete a Data Definition. . . . .	1250
TO2_deleteDS–Delete a Data Store . . . . .	1252
TO2_getBrowseDictPID–Get the Browser Dictionary PID . . . . .	1254
TO2_getClassAttributes–Get Attributes of a Class . . . . .	1256
TO2_getClassDocumentation–Copy Documentation for a Class . . . . .	1258
TO2_getClassInfo–Get Information for a Class . . . . .	1260
TO2_getClassNames–Convert EBCDIC Class Name to Index. . . . .	1262
TO2_getClassTree–Get an Inheritance Tree for a Class . . . . .	1264
TO2_getCollectionAttributes–Get the Attributes of the Collection . . . . .	1266
TO2_getCollectionName–Get the Class Name of a Collection. . . . .	1268
TO2_getCollectionParts–Get the Part Names of the Collection . . . . .	1270
TO2_getCreateTime–Copy Created Time Stamp of the Collection . . . . .	1272
TO2_getDDAttributes–Get the Current DD Attribute Values. . . . .	1274
TO2_getDirectoryForRRN–Get the Current Directory for the Specified RRN . . . . .	1276
TO2_getDSAttributes–Get the Attributes of the Data Store . . . . .	1278
TO2_getDSnameForPID–Determine the DS Name for a PID . . . . .	1280
TO2_getKeyPathAttributes–Retrieve the Attributes of Key Paths . . . . .	1281
TO2_getListDDnames–Retrieve the Defined Data Definition Names . . . . .	1283
TO2_getListDScollections–Retrieve the Data Store System Collections . . . . .	1285
TO2_getListDSnames–Retrieve the Defined Data Store Names . . . . .	1287
TO2_getListUsers–Get a List of Defined Users . . . . .	1289
TO2_getMethodDocumentation–Copy Documentation for a Method. . . . .	1291
TO2_getMethodNames–Get Method Names for a Class . . . . .	1293
TO2_getNumberOfRecords–Return the Number of Records Used . . . . .	1295
TO2_getPathInfoFor–Retrieve Path Information for a Collection Structure . . . . .	1296
TO2_getPIDforBrowseName–Retrieve the PID Associated with the Name . . . . .	1299
TO2_getPIDinventoryEntry–Get PID Inventory Entry for PID . . . . .	1301
TO2_getPIDinventoryPID–Get PID of PID Inventory of the Data Store. . . . .	1303
TO2_getRecordAttributes–Get Attributes of a Record File Address . . . . .	1305
TO2_getRecoupIndex–Get an Index . . . . .	1307
TO2_getRecoupIndexForPID–Retrieve Associated Recoup Index Name . . . . .	1310
TO2_getUserAttributes–Get User Attributes . . . . .	1312
TO2_isDDdefined–Test If DD Name Is Defined . . . . .	1314
TO2_isExtended–Internal Format of Collections . . . . .	1316
TO2_migrateCollection–Migrate a Collection . . . . .	1318



TO2_migrateDS—Migrate a Data Store . . . . .	1320
TO2_reclaimPID—Reclaim a PID . . . . .	1322
TO2_reconstructCollection—Reconstruct Collection . . . . .	1323
TO2_recoupCollection—Return Head of Chain File Addresses . . . . .	1325
TO2_recoupDS—Return a List of Internal PIDs to Be Recouped . . . . .	1327
TO2_recoupPT—Clear the Pool Reuse Table . . . . .	1329
TO2_recreateDS—Re-create a Data Store . . . . .	1330
TO2_removeBrowseKey—Remove the Element Using the Specified Key . . . . .	1332
TO2_restart—Restart TPF Collection Support at Cycle-Up . . . . .	1334
TO2_setGetTextDump—Set Text Dump On or Off . . . . .	1335
TO2_setMethodTrace—Set Method Trace On or Off . . . . .	1337
TO2_taskDispatch—Dispatch a TPF Collection Support Task . . . . .	1339
TO2_validateCollection—Cause a Collection to Be Validated . . . . .	1340
TO2_validateKeyPath—Cause a Key Path to Be Validated . . . . .	1342

<b>TPF Collection Support: Independent APIs . . . . .</b>	<b>1345</b>
TO2_class—Retrieve the User Class ID of the Collection . . . . .	1346
TO2_convertBinPIDtoEBCDIC—Make a PID Printable . . . . .	1348
TO2_convertEBCDICtoBinPID—Convert EBCDIC String to PID . . . . .	1350
TO2_deletePID—Delete a Persistent Identifier and Its Backing Store . . . . .	1352
TO2_setClass—Set the User Class ID of the Collection . . . . .	1354

<b>External Device Support . . . . .</b>	<b>1357</b>
Initial Setup . . . . .	1357
Writing Records . . . . .	1357
Retrieving Data . . . . .	1358
Error Codes . . . . .	1358
TPFxd_archiveEnd—Inform the Archive Facility That the Request Has Ended . . . . .	1360
TPFxd_archiveStart—Start the Archive Support Facility . . . . .	1361
TPFxd_close—Signal the End of a TPFxd_open Request . . . . .	1363
TPFxd_getPosition—Retrieve Current Positioning Information . . . . .	1365
TPFxd_getPrevPosition—Retrieve Previous Positioning Information . . . . .	1367
TPFxd_getVOLSER—Retrieve Current VOLSER and Media Type . . . . .	1369
TPFxd_getVOLSERlist—Retrieve a List of VOLSERs and Media Type . . . . .	1371
TPFxd_nextVolume—Advance to the Next Volume when Reading or Writing . . . . .	1372
TPFxd_open—Fulfill Any Mount or Positioning Required . . . . .	1374
TPFxd_read—Read from an External Device into a Buffer . . . . .	1376
TPFxd_readBlock—Read into a Core Block from an External Device . . . . .	1378
TPFxd_setPosition—Set Current Position of the External Device . . . . .	1380
TPFxd_sync—Verify Queued Information . . . . .	1382
TPFxd_write—Write to an External Device . . . . .	1384
TPFxd_writeBlock—Write Core Block Images to the External Device . . . . .	1386

<b>Appendix A. Conformance of TPF C Support to ANSI/ISO Standards . . . . .</b>	<b>1389</b>
Implementation-Defined Behavior of the TPF C Run-Time Library . . . . .	1389

<b>Appendix B. Standard C Header Files Supported by the TPF 4.1 System . . . . .</b>	<b>1393</b>
--	-------------

<b>Appendix C. Compiler Functions Supported as TPF 4.1 Extensions . . . . .</b>	<b>1395</b>
---	-------------

<b>Appendix D. C/C++ Functions Supported by the TPF 4.1 System but Not Documented . . . . .</b>	<b>1397</b>
Standard C/C++ Library Functions . . . . .	1397
XML Parser for C++ (XML4C) Version 3.5.1 Classes . . . . .	1402

<b>Appendix E. Programming Support for the TPF File System . . . . .</b>	<b>1405</b>
--	-------------

TPF File System C Functions . . . . .	1405
TPF-Supplied User and Group Names . . . . .	1410
Adding a User-Defined Device Driver . . . . .	1412
Special Files . . . . .	1413
Device Drivers and Commit Scopes . . . . .	1414
User-Defined Device Driver Functions . . . . .	1414
TPF_FSDD_APPEND–Write Beginning at the End of a File . . . . .	1416
TPF_FSDD_CLOSE–Close a File . . . . .	1419
TPF_FSDD_GET–Read From a File . . . . .	1420
TPF_FSDD_OPEN–Open a File . . . . .	1422
TPF_FSDD_POLL–Select or Poll an Open File Descriptor . . . . .	1425
TPF_FSDD_POLL_CLEAN–Select or Poll Cleanup of an Open File Descriptor . . . . .	1427
TPF_FSDD_PUT–Write to a File . . . . .	1429
TPF_FSDD_RESIZE–Change the Size of a File . . . . .	1432
TPF_FSDD_SIZE–Get the Size of a File . . . . .	1434
TPF_FSDD_SYNC–Synchronize the File Data . . . . .	1436
 <b>Appendix F. GNTAGH User’s Guide . . . . .</b>	 1437
C Global Tagnames . . . . .	1437
Program Parts List . . . . .	1437
Creating or Updating C Globals on VM . . . . .	1438
Creating or Updating C Globals on MVS . . . . .	1439
Format of Global Tags . . . . .	1441
GNTAGH Messages . . . . .	1442
GNTAGH Program Logic Flow . . . . .	1445
Sample Output . . . . .	1445
 <b>Appendix G. IPRSE - A Parser Utility for TPF Systems . . . . .</b>	 1447
Parser Options That Affect How the Input String Matches the Grammar . . . . .	1447
Special Characters for Input String Syntax . . . . .	1448
Defining a Grammar . . . . .	1448
IPRSE_parse–Parse a Text String against a Grammar . . . . .	1456
IPRSE_bldprstr–Initializing the Output Structure . . . . .	1466
 <b>Index . . . . .</b>	 1469

---

## Figures

1.	General Form for TPF/APPC Functions . . . . .	724
2.	File Descriptor and Open File Description . . . . .	1409
3.	Creating C Language Global Tagnames . . . . .	1441
4.	IPRSE_output Structure . . . . .	1459



# Tables

1. Modified Standard C Library Functions . . . . .	1
2. General Application Use C Library Functions . . . . .	2
3. Mapping TPF Assembler Services to C Functions . . . . .	3
4. Values for the Mode Parameter . . . . .	199
5. Flag Characters for the fprintf Family . . . . .	203
6. Precision Argument in the fprintf Family . . . . .	204
7. Type Characters and Their Meanings . . . . .	205
8. Conversion Specifiers in the fscanf and scanf functions . . . . .	219
9. getpc Error Return . . . . .	266
10. Priority Class Table (Shutdown Levels) . . . . .	310
11. relpc Error Return . . . . .	425
12. sigaction Function Specification Summary . . . . .	487
13. Signals Supported in the C/C++ Environment . . . . .	495
14. sonic Normal Return . . . . .	511
15. Elements of the stat Structure . . . . .	513
16. Ways of Exiting a TPF Process and the Resulting Exit Status . . . . .	529
17. tancc Normal Return . . . . .	530
18. tancc Error Return . . . . .	530
19. Time-Slice Name Table . . . . .	555
20. tpf_itrpc Normal Return . . . . .	608
21. tpf_itrpc Error Return . . . . .	608
22. tx_begin Normal Return . . . . .	647
23. tx_begin Error Return . . . . .	647
24. tx_commit Normal Return . . . . .	649
25. tx_commit Error Return . . . . .	649
26. tx_open Normal Return . . . . .	651
27. tx_open Error Return . . . . .	651
28. tx_resume_tpf Normal Return . . . . .	652
29. tx_resume_tpf Error Return . . . . .	652
30. tx_rollback Normal Return . . . . .	654
31. tx_rollback Error Return . . . . .	654
32. tx_resume_tpf Normal Return . . . . .	655
33. tx_resume_tpf Error Return . . . . .	655
34. uatbc Error Return . . . . .	657
35. wtopc_insert_header letter parameter . . . . .	704
36. TPF/APPC Basic Conversation Verbs and Valid Keywords . . . . .	725
37. Primary Return Codes . . . . .	731
38. Secondary Return Codes . . . . .	732
39. Return Codes When Conversation in SEND State . . . . .	783
40. Return Codes When Conversation in RECEIVE State . . . . .	783
41. Return Codes When Conversation in CONFIRM State . . . . .	784
42. Return Codes for TEST=POSTED . . . . .	786
43. Return Codes for TEST=RTSRCVD . . . . .	787
44. Return code when TEST is neither POSTED nor RTSRCVD . . . . .	787
45. TPF/APPC Mapped Conversation Return Codes . . . . .	796
46. Error Code Summary . . . . .	862
47. Collection Support: Non-Cursor APIs . . . . .	881
48. Collection Support: Cursor APIs . . . . .	1113
49. Standard C/C++ Library Functions Supported by the TPF System but Not Documented In This Book . . . . .	1397
50. TPF File System C Functions . . . . .	1405
51. TPF-Supplied User and Group Names . . . . .	1411
52. ACP.CSRCE.OL.RELv Data Set Members for Converting HLASM Output . . . . .	1437

53. Format of Global Tag Definitions . . . . .	1441
--	------

---

## Notices

References in this book to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service in this book is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
Department 830A  
Mail Drop P131  
2455 South Road  
Poughkeepsie, NY 12601-5400  
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Any pointers in this book to non-IBM Web sites are provided for convenience only and do not in any way serve as an endorsement. IBM accepts no responsibility for the content or use of non-IBM Web sites specifically mentioned in this book or accessed through an IBM Web site that is mentioned in this book.

---

## Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

- AD/Cycle
- AIX
- BookManager
- C/370
- DB2
- IBM
- Language Environment
- MQSeries
- MVS/ESA
- OS/2
- OS/390

- SP
- System/370
- System/390
- S/370
- VisualAge
- 3090
- 3890.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.



---

## About This Book

This book describes IBM C/C++ language support for Transaction Processing Facility (TPF) application programming, which permits application programmers to write TPF programs in C or C++ language. It is primarily a reference directed toward application programmers, although some functions will also be of interest to TPF system programmers and customer system and middleware programmers. Tools providers will also see a need for the information in this book. It serves as a reference guide to the library of C functions created for, or modified to work with, the TPF operating system. Use your compiler books at compile time, with the *TPF C/C++ Language Support User's Guide* and *TPF Application Programming*.

See *TPF Transmission Control Protocol/Internet Protocol* for information about the Transmission Control Protocol/Internet Protocol (TCP/IP) C functions.

Why code TPF application programs in C or C++? The benefits of writing TPF application programs in C or C++ language are significant. They include the following:

- Structured program logic makes programs easier to follow.
- Application programs can be easily modularized by taking advantage of the C *function* concept.
- Because C language statements are concise and the compiler does extensive syntax checking, fewer errors are introduced into application programs, increasing the probability that programs will run correctly the first time they are executed.
- Because an application programmer is not required to code registers explicitly, the likelihood of certain errors caused by corruption of main storage is greatly reduced.
- The level of TPF-specific knowledge required to develop applications is reduced.
- A large selection of programming tools is available to programmers in the form of C/C++ library functions.
- Storage for C variables is allocated by the system, reducing the need for programmers to manage data in core blocks.

All of these factors contribute to increased productivity in application programming groups.

IBM TPF C/C++ language support requires the use of the C/C++ compiler currently supported by IBM, which must be installed and available offline. Both the MVS and VM features of this compiler support the TPF system. C/C++ language support consists of modifications made to the TPF operating system to interface with IBM C/C++ compiler-generated code and a set of TPF-specific library functions.

In this book, abbreviations are often used instead of spelled-out terms. Every term is spelled out at first mention followed by the all-caps abbreviation enclosed in parentheses; for example, Systems Network Architecture (SNA). Abbreviations are defined again at various intervals throughout the book. In addition, the majority of abbreviations and their definitions are listed in the master glossary in the *TPF Library Guide*.

---

## Who Should Read This Book

This book is intended for application programmers who already understand some general TPF programming concepts, but who will be programming in C/C++ language rather than assembler.

The user of this book is expected to be a C/C++ language programmer who has some familiarity with TPF.

---

## Conventions Used in the TPF Library

The TPF library uses the following conventions:

Conventions	Examples of Usage
<i>italic</i>	Used for important words and phrases. For example: A <i>database</i> is a collection of data.  Used to represent variable information. For example: Enter <b>ZFRST STATUS MODULE</b> <i>mod</i> , where <i>mod</i> is the module for which you want status.
<b>bold</b>	Used to represent text that you type. For example: Enter <b>ZNALS HELP</b> to obtain help information for the ZNALS command.  Used to represent variable information in C language. For example: <b>level</b>
monospaced	Used for messages and information that displays on a screen. For example: PROCESSING COMPLETED  Used for C language functions. For example: maskc  Used for examples. For example: maskc(MASKC_ENABLE, MASKC_IO);
<b><i>bold italic</i></b>	Used for emphasis. For example: You <b><i>must</i></b> type this command exactly as shown.
<b><u>Bold underscore</u></b>	Used to indicate the default in a list of options. For example: <b>Keyword=OPTION1   <u>DEFAULT</u></b>
Vertical bar	Used to separate options in a list. (Also referred to as the OR symbol.) For example: <b>Keyword=Option1   Option2</b>  <b>Note:</b> Sometimes the vertical bar is used as a <i>pipe</i> (which allows you to pass the output of one process as input to another process). The library information will clearly explain whenever the vertical bar is used for this reason.
CAPital LETters	Used to indicate valid abbreviations for keywords. For example: KEYWord= <i>option</i>

Conventions	Examples of Usage
Scale	<p>Used to indicate the column location of input. The scale begins at column position 1. The plus sign (+) represents increments of 5 and the numerals represent increments of 10 on the scale. The first plus sign (+) represents column position 5; numeral 1 shows column position 10; numeral 2 shows column position 20 and so on. The following example shows the required text and column position for the image clear card.</p> <p> ...+....1....+....2....+....3....+....4....+....5....+....6....+....7...</p> <p>LOADER    IMAGE   CLEAR</p> <p><b>Notes:</b></p> <ol style="list-style-type: none"> <li>1. The word LOADER must begin in column 1.</li> <li>2. The word IMAGE must begin in column 10.</li> <li>3. The word CLEAR must begin in column 16.</li> </ol>

## Related Information

A list of related information follows. For information on how to order or access any of this information, call your IBM representative.

### IBM Transaction Processing Facility (TPF) 4.1 Books

- *TPF ACF/SNA Data Communications Reference*, SH31-0168
- *TPF ACF/SNA Network Generation*, SH31-0131
- *TPF Application Programming*, SH31-0132
- *TPF Application Requester User's Guide*, SH31-0133
- *TPF Concepts and Structures*, GH31-0139
- *TPF Database Reference*, SH31-0143
- *TPF General Macros*, SH31-0152
- *TPF Migration Guide: Program Update Tapes*, GH31-0187
- *TPF Operations*, SH31-0162
- *TPF System Installation Support Reference*, SH31-0149
- *TPF System Macros*, SH31-0151
- *TPF Transmission Control Protocol/Internet Protocol*, SH31-0120.

### IBM High-Level Language Books

- *C/C++ for MVS/ESA V3R2 Language Reference*, SC09-2150
- *C/C++ for MVS/ESA V3R2 Library Reference*, SC23-3881
- *C/C++ for MVS/ESA V3R2 Programming Guide*, SC09-2164
- *C/C++ for MVS/ESA V3R2 User's Guide*, SC09-2205
- *IBM C/370 Diagnosis Guide and Reference*, LY09-1804
- *IBM C/370 Programming Guide*, SC09-1384
- *IBM C/370 User's Guide*, SC09-1264
- *OS/390 C/C++ Language Reference*, SC09-2360
- *OS/390 C/C++ Programming Guide*, SC09-2362
- *OS/390 C/C++ Run-Time Library Reference*, SC28-1663
- *OS/390 C/C++ User's Guide*, SC09-2361
- *Programming Guide SAA AD/Cycle C/370*, SC09-1841
- *Programming Guide SAA AD/Cycle Language Environment/370*, SC09-1840

- *SAA AD/Cycle C/370 Language Reference*, SC09-1762
- *SAA AD/Cycle C/370 Library Reference*, SC09-1761
- *SAA AD/Cycle C/370 User's Guide*, SC09-1763
- *SAA Common Programming Interface C Reference - Level 2*, SC09-1308
- *Systems Application Architecture Common Programming Interface Communications Reference*, SC26-4399.

## IBM Message Queuing Books

- *MQSeries Application Programming Guide*, SC33-0807
- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries for AIX Application Programming Reference*, SC33-1374
- *MQSeries for MVS/ESA Application Programming Reference*, SC33-1212.
- *MQSeries for OS/2 Application Programming Reference*, SC33-1370
- *MQSeries Message Queue Interface Technical Reference*, SC33-0850.

## IBM Systems Network Architecture Books

- *IBM Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2*, SC30-3269
- *IBM Systems Network Architecture LU 6.2 Reference: Peer Protocols*, SC31-6808
- *IBM Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2*, GC30-3084.

## Miscellaneous IBM Books

- *3600 Finance Communication System 3614 Programmer's Guide and Reference*, GC27-0010.

## Online Information

- *Messages (Online)*
- *Messages (System Error and Offline)*
- *XML User's Guide.*

---

## How to Send Your Comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other TPF information, use one of the methods that follow. Make sure you include the title and number of the book, the version of your product and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

- If you prefer to send your comments electronically, do either of the following:
  - Go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>.  
There you will find a link to a feedback page where you can enter and submit comments.
  - Send your comments by e-mail to [tpfid@us.ibm.com](mailto:tpfid@us.ibm.com)

- If you prefer to send your comments by mail, address your comments to:

IBM Corporation  
TPF Systems Information Development  
Mail Station P923  
2455 South Road  
Poughkeepsie, NY 12601-5400  
USA

- If you prefer to send your comments by FAX, use this number:
  - United States and Canada: 1 + 845 + 432 + 9788
  - Other countries: (international code) + 845 + 432 +9788



---

# TPF API Functions

This chapter contains an alphabetic listing of most TPF API functions. There are additional sets of functions that provide the interface for TPF Advanced Program-to-Program Communications (TPF/APPC). The descriptions for these functions are provided in the following chapters:

- “TPF/APPC Basic Conversation Functions” on page 723
- “TPF/APPC Mapped Conversation Functions” on page 793.

You can find more information about TPF/APPC support in *TPF ACF/SNA Data Communications Reference*.

There is also an additional set of functions for TCP/IP support. These functions are described in *TPF Transmission Control Protocol/Internet Protocol*.

The following information is shown for each TPF API function:

**Format** The function prototype and a description of any parameters.

**Description** The service that the function provides.

**Normal Return** What is returned when the requested service has been performed.

**Error Return** What is returned when the requested service could not be performed. Specifying incorrect function parameters results in a system error with exit.

**Programming Considerations** Remarks that help the programmer to understand the proper use of the function and any side effects that may occur when the function is executed. Also, if the use of a particular function affects the use of another function, that is described.

**Example** A code segment, showing a sample function call.

**Related Functions** Where to find additional information pertinent to this function.

**Note:** Remember to include `tpfeq.h` in the TARGET(TPF) source program even though it is not shown in any of the function prototypes. The `tpfeq.h` header must be the first header file included in any TPF C source module.

Several standard C library functions have been rewritten or substantially modified to run in the TPF system environment. For this reason, they are included in this reference section, even though they operate in the standard fashion; formats and return values have not been changed. These modified standard C library functions include:

*Table 1. Modified Standard C Library Functions*

Function	Description
abort	Terminate program abnormally
assert	Verify condition or print diagnostic message
exit	Exit an ECB
perror	Write error message to standard error stream
fprintf	Format and write data

Table 1. Modified Standard C Library Functions (continued)

Function	Description
printf	Format and write data
sprintf	Format and write data
scanf	Scan input for variables

The remaining functions described in this chapter may be thought of as being on two levels; one group of them is better suited to *general application use* for one or more of the following reasons:

- One function call results in several TPF macro service routines being performed.
- The return information is more meaningful.
- Much of the system-specific data manipulation is done internally.

The functions we are talking about include the following:

Table 2. General Application Use C Library Functions

Function	Description
file_record	File a record
find_record	Find a record
global	Address or operate on TPF globals
glob	Address global field or record
glob_keypoint	Keypoint a global field or record
glob_lock	Lock and Access a synchronizable global field or record
glob_modify	Modify a global field or record
glob_sync	Synchronize a global field or record
glob_unlock	Unlock a global field or record
glob_update	Update a global field or record
levtest	Test core level for occupied condition
tape_close	Close a general tape
tape_cntl	Tape control
tape_open	Open a general tape
tape_read	Read a record from general tape
tape_write	Write a record to general tape

These functions consist of a subset of system services.

The other group of functions, that are not as well suited to general application use, correspond to TPF macros written in assembler. Each of these functions performs one operation. They give the C programmer more control, direct access to the ECB (and registers), and assume that the programmer is coding them in the proper sequence. Some installations may want to restrict the use of these functions.

You may want to extend the existing set of higher-level functions to create a complete set, which includes the remaining system services. This can be done by adding a #define statement to the following functions and supplying alternative names for the preprocessor:



Function	Description	Suggested Alternative Name
getcc	Obtain working storage block	getcore
getfc	Obtain file pool address	gfs
relcc	Release working storage block	relcore
relfc	Release file pool storage	rfs
unfrc	Unhold a file record	unhold
waitc	Wait for outstanding I/O completion	wait

The TPF control program does not support C language programs. The implementation of C requires ECB virtual memory (EVM) and register connection to an ECB.

## Mapping Assembler Macros to C Functions

The following section is provided for those programmers who are making the transition from TPF application programming in assembler, to C language. Many assembler macros do not have an equivalent in the C language. This is because they are not relevant to C language programming, or there are applicable existing C library functions. Conversely, there are many C functions that have no assembler equivalent.

The entries that appear in the Assembler Service column include macros, as well as program segments that are commonly entered to perform certain common system functions, such as date conversion and message transmission.

The following table lists the C functions that perform equivalent services to their assembler counterparts, either in whole or in part.

*Table 3. Mapping TPF Assembler Services to C Functions*

Assembler Service	Corresponding C Functions
ADDLC	addlc
ALPHA	scanf, sscanf
ATTAC	attac, attac_ext
CALOC	calloc
CEBIC	cebic_save, cebic_restore, cebic_goto_bss, cebic_goto_dbi, cebic_goto_ssu
CENVC	getenv, setenv, unsetenv
CIFRC	cifrc
CINFC	cinfo, cinfo_fast, cinfo_fast_ss
CORHC	corhc
CORUC	coruc
CRASC	printf, routc
CRATC	cratc
CREDC	credc
CREEC	creec

Table 3. Mapping TPF Assembler Services to C Functions (continued)

Assembler Service	Corresponding C Functions
CREMC	cremc
CRESC	tpf_cresc
CRETC	cretc, cretc_level
CREXC	crexc
CROSC	crosc_entrc
CRUSA	crusa, relcc
CSOnc	csonc
DBSAC	dbzac
DBSDC	dbzdc.h
DECBC	tpf_decb_create, tpf_decb_locate, tpf_decb_release, tpf_decb_swapblk, tpf_decb_validate
DEFRC	defrc
DEQC	deqc
DETAC	detac, detac_ext
DLAYC	dlayc
EDITA	scanf, sscanf
ENQC	enqc
ENTDC	entdc
ENTRC	entrc, TPF_CALL_BY_NAME, normal external function call.
ESFAC	tpf_esfac
EVINC	evinc
EVNQC	evnqc
EVNTC	evntc
EVNWC	evnwc
EXITC	abort, exit
FA4X4C	tpf_fa4x4c
FAC8C	tpf_fac8c
FACZC	tpf_faczc
FILEC	file_record, file_record_ext, filec, filec_ext
FILKW	global
FILNC	filnc, filnc_ext, file_record_ext
FILUC	file_record, filuc, filuc_ext, file_record_ext
FINDC	findc, findc_ext, find_record_ext
FINHC	find_record, finhc, finhc_ext, find_record_ext
FINWC	find_record, finwc, finwc_ext, find_record_ext
FIWHC	find_record, fiwhc, fiwhc_ext, find_record_ext
FLIPC	flipc
FMSG	printf, puts, sprintf
FREEC	free

Table 3. Mapping TPF Assembler Services to C Functions (continued)

Assembler Service	Corresponding C Functions
GDSNC	gdsnc
GDSRC	gdsrc
GETCC	getcc
GETFC	getfc
GETPC	getpc
GLOBY	glob, global
GLMOD	global
GLOBZ	glob, global
GLOUC	global, glob_keypoint, glob_update
INQRC	inqrc
KEYRC	keyrc
LEVTA	levtest
\$LOCKC	lockc
LODIC	lodic, lodic_ext
LONGC	longc
MALOC	malloc
MOVEC	tpf_movec
PAUSC	pausc
PROGC	progc
POSTC	postc
RAISA	raisa
RALOC	realloc
RCRFC	tpf_rcrfc
RCUNC	rcunc
REHKA	rehka
RELCC	relcc
RELFC	relfc
RELPC	relpc
RIDCC	ridcc
RLCHA	rlcha
ROUTC	printf, puts, routc
RVTCC	rvtcc
SELEC	selec
SENDC	printf, puts, routc
SERRC	serrc_op
SNAPC	snapc
SONIC	sonic
SYNCC	global, glob, glob_sync, glob_update
SYSTC	systc

Table 3. Mapping TPF Assembler Services to C Functions (continued)

Assembler Service	Corresponding C Functions
TASNC	tasnc
TANCC	tancc
TBSPC	tape_cntl, tbspc
TCLSC	tape_close, tclsc
TDSPC	tdspc, tdspc_q
TMCNA	ctime, time, gmtime
TMSLC	tmslc
TOPNC	tape_open, topnc
TOURC	tourc
TOUTC	toutc
TPCNC	tape_cntl
TPPCC	tppc_activate_on_confirmation, tppc_activate_on_receipt, tppc_allocate, tppc_confirm, tppc_confirmed, tppc_deallocate, tppc_flush, tppc_get_attributes, tppc_get_type, tppc_post_on_receipt, tppc_prepare_to_receive, tppc_receive, tppc_request_to_send, tppc_send_data, tppc_send_error, tppc_test, tppc_wait
TPRDC	tape_read, tprdc
TREWC	tape_cntl, trewc
TR SVC	trsvc
TSYNC	tape_close, tsync
TWRTC	tape_write, twrtc
TXBGC	tx_begin
TXCMC	tx_commit
TXRBC	tx_rollback
TXRSC	tx_resume_tpf
TXSPC	tx_suspend_tpf
TYCVA	ctime, time, gmtime
UATBC	uatbc
UCDR	gmtime, ctime
UIO1	puts
UNFRC	unfrc, unfrc_ext
UNHKA	unhka
\$UNLKC	unlkc
WAITC	waitc
WGTAC	wgtac
WTOPC	wtopc, wtopc_text

Function prototypes are found in header files. Prototypes for the TPF API functions are normally found in `tpfapi.h`. TPF find and file function prototypes are found in `tpfio.h`, while the C language I/O function prototypes are in `stdio.h`.

---

## abort—Terminate Program Abnormally

This function stops a program.

### Format

```
#include <stdlib.h>
void abort(void);
```

This function causes an abnormal program termination and returns control to the host environment. No dumps are issued for entry control block (ECB) states that are not valid such as having a hold on a file address, having a general tape open or assigned, or having issued a PAUSC macro to place the system in a uniprocessor environment (these conditions are cleared by exit processing).

If the abort function is called and the user has a handler for SIGABRT, then SIGABRT is raised; however, SIGABRT is raised again with the default handler if the user's handler returns, even if it has set another SIGABRT handler. The code path only passes through a user handler once, even if the handler is reset. Abnormal termination also occurs if SIGABRT is ignored.

**Note:** Calling the abort function will not result in program termination if SIGABRT is caught by a signal handler, and the signal handler does not return. You can avoid returning by *jumping* out of the handler with the setjmp and longjmp functions.

As part of termination, the system attempts to complete normal exit processing, including calling atexit functions, flushing and closing open files, and removing files created by tmpfile. If the attempt at normal termination causes a system error, the process is immediately forced to exit.

The exit status that is returned to a parent process depends on how the SIGABRT handler causes the process to exit.

#### TARGET(TPF) restriction

TARGET(TPF) does not support the raise function; therefore, the TARGET(TPF) version of abort does not raise SIGABRT. The status returned to a parent process is undefined.

### Normal Return

The abort function does not return to its caller and causes abnormal termination of the calling process.

### Error Return

Not applicable.

### Programming Considerations

- Following a call to the abort function, no return is made to the operational program.
- If the abort function is called within a commit scope, processing ends as if a rollback was issued.

## abort

### Examples

The following example tests for successful opening of the file `myfile`. If an error occurs, an error message is printed and the program ends with a call to the `abort` function.

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

int main(void)
{
    FILE *stream;

    if ((stream = fopen("myfile.dat", "r")) == NULL)
    {
        printf("Could not open data file\n");
        abort();

        printf("Should not see this message\n");
    }
}
```

### Related Information

- “`assert`—Verify Condition or Print Diagnostic Message” on page 15
- “`exit`—Exit an ECB” on page 115
- “`signal`—Install Signal Handler” on page 495.

## access—Determine Whether a File Can Be Accessed

This function determines how the requesting process can access a file. When checking to see if a process has appropriate permissions, the access function looks at the real user ID (UID) and real group ID (GID) of the process, not the effective UID and effective GID.

### Format

```
#include <unistd.h>
int access(const char *pathname, int how);
```

#### **pathname**

The name of the file whose accessibility you want to test.

#### **how**

Indicates the access modes you want to test.

The following symbols are defined in the `unistd.h` header file for use in the **how** parameter:

#### **F\_OK**

Tests whether the file exists.

#### **R\_OK**

Tests whether the process can read the file.

#### **W\_OK**

Tests whether the process can write the file.

#### **X\_OK**

Tests whether the process can search the directory.

You can take the bitwise inclusive OR of any or all of the last three symbols to test several access modes at once. If you are using `F_OK` to test for the existence of a file, you cannot use OR with any of the other symbols.

### Normal Return

The returned value of access is zero if the specified access is permitted.

### Error Return

If unsuccessful, the access function returns `-1` and sets `errno` to one of the following:

#### **EACCES**

The process does not have appropriate permissions to access the file in the specified way or does not have search permission on some component of the **pathname** prefix.

#### **EINVAL**

The value of **how** is incorrect.

#### **ELOOP**

A loop exists in the symbolic links. This error is issued if the number of symbolic links detected in the resolution is greater than `POSIX_SYMLINK` (a value defined in the `limits.h` header file).

#### **ENAMETOOLONG**

**pathname** is longer than `PATH_MAX` characters, or some component of **pathname** is longer than `NAME_MAX`.

#### **ENOENT**

There is no file named **pathname** or the **pathname** parameter is an empty string.

#### **ENOTDIR**

Some component of the **pathname** prefix is not a directory.

## access

### Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

### Examples

The following example determines how a file is accessed.

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

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

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

#### Output

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

### Related Information

“chmod—Change the Mode of a File or Directory” on page 32.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



## addlc—Add a Block to the TPF Task Dispatch List

This function adds a block to a specified TPF system task dispatch list.

### Format

```
#include <sysapi.h>
void addlc(void *block,
           enum addlc_addr_space block_address_space,
           enum addlc_list list_name,
           enum addlc_position where_to_add,
           void *post_interrupt_address);
```

#### **block**

A pointer to a TPF storage block.

#### **block\_address\_space**

Whether the block address is an EVM or SVM address. Valid values for this parameter are:

##### **ADDLC\_BLK\_EVM**

The block address is an EVM address.

##### **ADDLC\_BLK\_SVM**

The block address is an SVM address.

#### **list\_name**

The name of the task dispatch list to which the block will be added. Valid values for this parameter are:

##### **ADDLC\_READY**

The block will be added to the ready list.

##### **ADDLC\_INPUT**

The block will be added to the input list.

##### **ADDLC\_DEFER**

The block will be added to the defer list.

#### **where\_to\_add**

Where in the list the block will be added. Valid values for this parameter are:

##### **ADDLC\_TOP**

The block will be added to the top of the task dispatch list.

##### **ADDLC\_BOTTOM**

The block will be added to the bottom of the task dispatch list.

#### **post\_interrupt\_address**

Pointer to the post-interrupt address.

### Normal Return

Void.

### Error Return

None.

### Programming Considerations

- The addlc function disconnects the block from the ECB private area (EPA) if **block\_address\_space** is **ADDLC\_EVM**. On return, the program must clear the block held indicator from the core block reference word (CBRW) to prevent double release problems in EXITC processing.

## **addlc**

### **Examples**

The following example adds a block to the top of the ready list, assuming `ebw000` has the block SVM address and `ebw004` has the post-interrupt address:

```
#include <sysapi.h>
addlc(*(void **)&ecbptr()->ebw000,
      ADDLC_BLK_SVM,
      ADDLC_READY,
      ADDLC_TOP,
      *(void **)&ecbptr()->ebw004);
```

### **Related Information**

None.

## alarm—Schedule an Alarm

This function causes the TPF system to send the calling process a SIGALRM signal after a specified amount of time has elapsed.

### Format

```
#include <unistd.h>
unsigned int alarm(unsigned int time_value);
```

#### time\_value

Specifies, in seconds, the number of time units until SIGALRM is generated. The maximum time value is 65 535. A higher value can be specified, but a value of 65 535 will actually be used. A value of zero causes an outstanding alarm request to be canceled and the SIGALRM signal will not be sent.

### Normal Return

The alarm function is always successful.

If there is a previous alarm request with time remaining, the alarm function returns the number of seconds until the previous request would have generated a SIGALRM signal. If the remaining time includes a fraction of a second, it is rounded up to the next second in the return value. Otherwise, the alarm function returns zero.

### Error Return

Not applicable.

## Programming Considerations

Alarm requests are not stacked. If the SIGALRM signal has not yet been generated, the call results in rescheduling the time when SIGALRM will be generated.

## Examples

The following example shows how to set an alarm.

```
#include <unistd.h>
unsigned int my_alarm = 3; /* three seconds */

:

/* Set alarm before entering select statement*/
rc = alarm(my_alarm);

:
```

## Related Information

- “kill—Send a Signal to a Process” on page 298
- “sigaction—Examine and Change Signal Action” on page 486
- “signal—Install Signal Handler” on page 495
- “sigpending—Examine Pending Signals” on page 498
- “sigprocmask—Examine and Change Blocked Signals” on page 499
- “sigsuspend—Set Signal Mask and Wait for a Signal” on page 502
- “sleep—Suspend the Calling Process” on page 507
- “tpf\_fork—Create a Child Process” on page 594
- “tpf\_process\_signals—Process Outstanding Signals” on page 622

## alarm

- “wait—Wait for Status Information from a Child Process” on page 684
- “waitpid—Obtain Status Information from a Child Process” on page 687.

## assert—Verify Condition or Print Diagnostic Message

This function, which is implemented as a macro, either verifies a condition or else prints a diagnostic message and abnormally ends the program.

### Format

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

#### **expression**

Any valid C logical expression to control the action of the assert macro.

The assert macro inserts conditional diagnostics into a program. If **expression** is true (nonzero), the assert macro suppresses the inserted diagnostics. Otherwise, the assert macro prints the diagnostic message and calls the abort function to abnormally end the program.

The diagnostic message has the following format:

Assertion failed: *expression*, file: *filename*, line: *line-number*

The assert macro uses the `__FILE__` and `__LINE__` predefined names to generate the printed diagnostic message.

### Normal Return

None.

### Error Return

None.

## Programming Considerations

- Using the `#undef` C preprocessor directive with the assert macro results in undefined behavior.
- If the program or any of its header files has defined **NDEBUG** by the time the `<assert>` header is included, the C preprocessor expands the assert macro to a void expression rather than to the **expression** to control the assert action. Any side effects associated with the **expression** to control the assert action only occur when **NDEBUG** is *not* defined.
- Beware that a program can have alternative behaviors depending upon whether **NDEBUG** is defined or not. Avoid these alternative behaviors by avoiding side effects in the **expression** used to control the assert action. The **expression** to control the assert action should not have any side effects because that **expression** is not always compiled. That **expression** is only compiled when **NDEBUG** is *not* defined.

#### **TARGET(TPF) restriction**

For TARGET(TPF), the assert macro relies on the puts function in order to operate. If assert is called and puts has not been implemented, control is passed to the system error routine.

## assert

### Examples

In the following example, the `assert` macro tests the string parameter for a null string and an empty string, and verifies that the length parameter is positive before proceeding.

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

void analyze(char *, int);
int main(void)
{
    char *string1 = "ABC";
    char *string2 = "";
    int length = 3;

    analyze(string1, length);
    printf("string1 %s is not null or empty, "
           "and has length %d \n", string1, length);
    analyze(string2, length);
    printf("string2 %s is not null or empty,"
           "and has length %d\n", string2, length);
}

void analyze(char *string, int length)
{
    assert(string != NULL);           /* cannot be NULL */
    assert(*string != '\0');         /* cannot be empty */
    assert(length > 0);              /* must be positive */
}
```

#### Output

```
String1 ABC is not null or empty, and has length 3
Assertion failed: *string != '\0',
file: CBC3BA08 C      A1, line: 26
```

### Related Information

“abort—Terminate Program Abnormally” on page 7.

## atexit—Call a Function at Normal Program Termination

The `atexit` function records a function, pointed to by **func** that the TPF system calls at *normal program termination*. Normal program termination is the result of calling the `exit` function, returning from the `main` function, or calling the Basic Assembler Language (BAL) `EXITC` macro.

You can use the `atexit` function to register as many as 32 functions. The registered functions are called in reverse order. The registered functions must return to ensure that all registered functions are called.

### Format

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

#### **func**

A pointer to a function to be called at normal program termination.

### Normal Return

A zero value indicates that the function has completed successfully.

### Error Return

A nonzero value indicates that the function has not completed successfully.

## Programming Considerations

- The `atexit` function does not support registering function pointers returned from other C load modules. The TPF system does not support the use of a function pointer returned from another C load module. You can use `atexit` to register library functions, dynamic load modules (DLMs), or functions that are contained in the same C load module as the call to `atexit`.
- The `atexit` function handles writable static for functions that are registered in C load modules other than the one that causes the entry control block (ECB) to exit.
- If the `atexit` function is called in a C load module other than the one that causes the ECB to exit, ensure that the C load module containing the registered function is still in main storage when the ECB exits.
- If the ECB runs on more than one I-stream or in more than one subsystem, functions registered by `atexit` may run on an I-stream or subsystem different from the one where they were registered. The registered function must ensure that it can run correctly when the process ends normally.

For example, if a process that runs on more than one I-stream calls `atexit` to register a function that accesses I-stream unique globals, the registered function must determine if it is running on the required I-stream and, if it is not, either switch itself to the correct I-stream or return without accessing the I-stream unique globals.

- It is not possible to *unregister* a function that has previously been registered by calling `atexit`.
- A registered function must take into account any conditions at normal program termination that can affect its ability to run.
- If multiple `atexit` functions are registered and one does not exit normally, the remaining `atexit` functions will not be called.

## atexit

### Examples

The following example uses the `atexit` function to call the `goodbye()` function at normal program termination.

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

void goodbye(void);

int main(void)
{
    int rc;

    rc = atexit(goodbye);
    if (rc != 0)
        puts("Error in atexit.");

    return 0;
}

void goodbye(void)
{
    /* This function is called at normal program termination. */
    puts("The goodbye() function was called at program termination.");
}
```

#### Output

The `goodbye()` function was called at program termination.

### Related Information

“exit–Exit an ECB” on page 115.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



## attac—Attach a Detached Working Storage Block

The **attac** function reattaches a working storage block to an ECB. The ECB must not be holding a storage block on the specified level — it must have been previously released using the detach functions.

### Format

```
#include <tpfapi.h>
void      *attac(enum t_lvl level);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this core block reference word (CBRW) level is the block to be attached.

### Normal Return

The pointer of type `void` representing the address of the start of the newly attached block.

### Error Return

Not applicable.

## Programming Considerations

- A system error with exit results if an incorrect data level is specified, if no storage block was previously detached on the referenced level, or when there is already a block on the specified level.
- Excessive use of this function can cause a depletion of working storage; therefore, its use should be carefully monitored.
- The detach functions (`detach`, `detach_ext`, and `detach_id`) and the attach functions (`attac`, `attac_ext`, and `attac_id`) push and pop working storage blocks to and from a list chained from the ECB. The `attac` function always attaches the block most recently detached from the specified data level. The `attac_ext` function attaches different blocks depending on how it is coded. The `attac_id` function attaches a working storage block with a matching FARW field to an ECB.  
When coding attach and detach function calls, use the `attac` and `detach` function calls together, or the `attac_ext` and `detach_ext` function calls together, or the `attac_id` and `detach_id` function calls together.
- The specified data level CBRW, FARW, and FARW extension are restored to the contents at the time the most recent `detach` function was called for that data level.

## Examples

The following example reattaches a previously detached IM0IM record to level D6.

```
#include <tpfapi.h>
struct im0im *inm;
:
:
inm = (struct im0im *)attac(D6);
```

## Related Information

- “getcc—Obtain Working Storage Block” on page 248
- “detach—Detach a Working Storage Block from the ECB” on page 85
- “attac\_ext—Attach a Detached Working Storage Block” on page 21

## **attac**

- “detac\_ext–Detach a Working Storage Block from the ECB” on page 86
- “attac\_id–Attach a Detached Working Storage Block” on page 23
- “detac\_id–Detach a Working Storage Block from the ECB” on page 88.

## attac\_ext—Attach a Detached Working Storage Block

This function reattaches a working storage block to an entry control block (ECB) data level or data event control block (DECB). The ECB must not be holding a storage block on the specified ECB data level or DECB; it must have been previously released using the `detac_ext` function.

### Format

```
#include <tpfapi.h>
void      *attac_ext(enum t_lvl level, int ext);
```

or

```
#include <tpfapi.h>
void      *attac_ext(TPF_DECB *decb, int ext);
```

#### level

One of 16 possible values representing a valid ECB data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the ECB data level (0–F). The working storage block on this core block reference word (CBRW) level is the block to be attached.

#### decb

A pointer to a DECB.

#### ext

Use one of following choices to indicate which block is to be attached.

##### ATTAC\_USER\_DEFAULT

Indicates the user is unspecified, which implies that the last block detached from this ECB data level should now be attached.

##### ATTAC\_USER\_ACPDB

Indicates the user is ACPDB, which implies that the block to be attached is the previously detached block that has a matching file address reference word (FARW) key. The caller of this function must set up the FARW for the ECB data level or DECB specified in the `attac_ext` function call to contain the same FARW contents as when the block was detached through the `detac_ext` function call. The service routine will attach the block that has a matching key; the key is the contents of the FARW for the specified ECB data level or DECB.

### Normal Return

Pointer of type `void` representing the address of the start of the newly attached block.

### Error Return

Not applicable.

### Programming Considerations

- A system error with exit results if an incorrect ECB data level or DECB is specified, or if no storage block was previously detached on the referenced ECB data level or DECB.
- Excessive use of this function can cause a depletion of working storage; therefore, its use should be carefully monitored.
- The detach functions (`detac`, `detac_ext`, and `detac_id`) and the attach functions (`attac`, `attac_ext`, and `attac_id`) **push** and **pop** working storage blocks to and

## attac\_ext

from a list chained from the ECB. The `attac` function always attaches the block most recently detached from the specified data level. The `attac_ext` function attaches different blocks depending on how it is coded. The `attac_id` function attaches a working storage block with a matching FARW field to an ECB.

When coding attach and detach function calls, use the `attac` and `detac` function calls together, or the `attac_ext` and `detac_ext` function calls together, or the `attac_id` and `detac_id` function calls together.

- The specified ECB data level or DECB CBRW, FARW, and FARW extension are restored to the contents at the time the most recent `detac` or `detac_ext` function was called for that ECB data level or DECB.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example reattaches a previously detached IM0IM record to level D6.

```
#include <tpfapi.h>
struct im0im *inm;
:
inm = (struct im0im *)attac_ext(D6,ATTAC_USER_ACPDB);
```

The following example reattaches a previously detached IM0IM record to the DECB referenced by variable `decb`.

```
#include <tpfapi.h>
struct im0im *inm;
TPF_DECB *decb;
:
inm = (struct im0im *)attac_ext(decb,ATTAC_USER_ACPDB);
```

## Related Information

- “`attac`—Attach a Detached Working Storage Block” on page 19
- “`detac`—Detach a Working Storage Block from the ECB” on page 85
- “`detac_ext`—Detach a Working Storage Block from the ECB” on page 86
- “`attac_id`—Attach a Detached Working Storage Block” on page 23
- “`detac_id`—Detach a Working Storage Block from the ECB” on page 88
- “`getcc`—Obtain Working Storage Block” on page 248.

See *TPF Application Programming* for more information about DECBs.

## attac\_id—Attach a Detached Working Storage Block

This function reattaches a working storage block to an entry control block (ECB) data level or data event control block (DECB). The ECB must not be holding a storage block on the specified ECB data level or DECB; it must have been previously released using the `detac_id` function.

The block to be attached is the previously detached block that has a matching file address reference word (FARW) key. The caller of this function must set up the FARW for the ECB data level or DECB specified in the `attac_id` function call to contain the same FARW contents as when the block was detached through the `detac_id` function call.

### Format

```
#include <tpfapi.h>
void      *attac_id(enum t_lvl level);
```

or

```
#include <tpfapi.h>
void      *attac_id(TPF_DECB *decb);
```

#### level

One of 16 possible values representing a valid ECB data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this core block reference word (CBRW) level is the block to be attached.

#### decb

A pointer to a DECB.

### Normal Return

Void. The pointer of type `void` represents the address of the start of the newly attached block.

### Error Return

Not applicable.

### Programming Considerations

- A system error with exit results if an incorrect data level or DECB is specified, or if no storage block was previously detached on the referenced ECB data level or DECB.
- Excessive use of this function can cause a depletion of working storage; therefore, its use should be carefully monitored.
- The detach functions (`detac`, `detac_ext`, and `detac_id`) and the attach functions (`attac`, `attac_ext`, `attac_id`) **push** and **pop** working storage blocks to and from a list chained from the ECB. The `attac` function always attaches the block most recently detached from the specified ECB data level. The `attac_ext` function attaches different blocks depending on how it is coded. The `attac_id` function attaches a working storage block with a matching FARW field to an ECB.

When you code attach and detach function calls, use the `attac` and `detac` function calls together, or the `attac_ext` and `detac_ext` function calls together, or the `attac_id` and `detac_id` function calls together.

## attac\_id

- The specified ECB data level or DECB CBRW, FARW, and FARW extension are restored to the contents at the time the most recent `detac` or `detac_ext` function was called for that ECB data level or DECB.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.

## Examples

The following example reattaches a previously detached IM0IM record to level D6.

```
#include <tpfapi.h>
struct im0im *inm;
:
:
inm = (struct im0im *)attac_id(D6);
```

The following example reattaches a previously detached IM0IM record to the DECB referenced by variable `decb`.

```
#include <tpfapi.h>
struct im0im *inm;
TPF_DECB *decb;
:
:
inm = (struct im0im *)attac_id(decb);
```

## Related Information

- “attac—Attach a Detached Working Storage Block” on page 19
- “detac—Detach a Working Storage Block from the ECB” on page 85
- “attac\_ext—Attach a Detached Working Storage Block” on page 21
- “detac\_ext—Detach a Working Storage Block from the ECB” on page 86
- “detac\_id—Detach a Working Storage Block from the ECB” on page 88
- “getcc—Obtain Working Storage Block” on page 248.

See *TPF Application Programming* for more information about DECBs.

## cebic\_goto\_bss—Change MDBF Subsystem to BSS

This function changes the MDBF subsystem to the BSS.

### Format

```
#include <sysapi.h>
void      cebic_goto_bss(void);
```

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

The MDBF subsystem is changed to BSS.

## Examples

The following example changes the DBI and SSU ID to which the ECB belongs.

```
#include <sysapi.h>
#include <tpfapi.h>
#include <string.h>

struct uatbc_area  uatbc_parm;

uatbc_parm.uat_call = UAT_NSSI;
memcpy(&uatbc_parm.uat_parm, "MYSS", 4);

if (uatbc(&uatbc_parm) == UAT_SUCCESS)
{
    struct msOut    *ssu_slot_ptr = uatbc_parm.uat_ssu_ptr;
    unsigned short  new_dbi = *(unsigned short *)ssu_slot_ptr->mu0dpi;

    cebic_goto_dbi(new_dbi);

    :
}
```

## Related Information

- “cebic\_goto\_dbi—Change MDBF Subsystem to DBI” on page 26
- “cebic\_goto\_ssu—Change MDBF Subsystem” on page 27
- “cebic\_restore—Restore Previously Saved DBI and SSU IDs” on page 28
- “cebic\_save—Save the Current DBI and SSU IDs” on page 29.

## cebic\_goto\_dbi—Change MDBF Subsystem to DBI

This function changes the MDBF subsystem to the specified DBI. The subsystem user is changed to the first in the specified DBI.

### Format

```
#include <sysapi.h>
void      cebic_goto_dbi(unsigned short int dbi);
```

#### dbi

The 2-byte MDBF ID of the SS to change to.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example changes the DBI and SSU ID to which the ECB belongs.

```
#include <sysapi.h>
#include <c$uatbc.h>
#include <c$msOut.h>
    unsigned short new_dbi;
    struct uatbc_area uatbc_parm;
    struct msOut *ssu_slot_ptr;
    :
    uatbc_parm.uat_call = UAT_NSSI;
    memcpy (&uatbc_parm.uat_parm, "MYSS", 4); /* SS name */
    if (uatbc(&uatbc_parm) == 0)
    {
        ssu_slot_ptr = uatbc_parm.uat_ssu_ptr; /* pointer to
                                                subsystem user slot */
        :
        new_dbi = *(unsigned short *) ssu_slot_ptr->mu0dpi;
        cebic_goto_dbi(new_dbi);
    }
```

### Related Information

- “cebic\_goto\_bss—Change MDBF Subsystem to BSS” on page 25
- “cebic\_goto\_ssu—Change MDBF Subsystem” on page 27
- “cebic\_restore—Restore Previously Saved DBI and SSU IDs” on page 28
- “cebic\_save—Save the Current DBI and SSU IDs” on page 29
- “uatbc—MDBF User Attribute Reference Request” on page 657.



## cebic\_goto\_ssu—Change MDBF Subsystem

This function changes the MDBF subsystem user to the specified SSU. If the specified subsystem user does not belong to the current subsystem, the DBI is changed to the parent of the new subsystem user.

### Format

```
#include <sysapi.h>
void      cebic_goto_ssu(unsigned short int ssu);
```

#### **ssu**

The 2-byte MDBF ID of the SSU to change to.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example changes the SSU ID to which the ECB belongs. If the new SSU belongs to a different SS than the existing one, the DBI is also changed.

```
#include <sysapi.h>
unsigned short new_ssu;
:
cebic_goto_ssu(new_ssu);
```

### Related Information

- “cebic\_goto\_bss—Change MDBF Subsystem to BSS” on page 25
- “cebic\_goto\_dbi—Change MDBF Subsystem to DBI” on page 26
- “cebic\_restore—Restore Previously Saved DBI and SSU IDs” on page 28
- “cebic\_save—Save the Current DBI and SSU IDs” on page 29.

## cebic\_restore—Restore Previously Saved DBI and SSU IDs

This restores the DBI and SSU IDs last saved by the existing ECB.

### Format

```
#include <sysapi.h>
void      cebic_restore(void);
```

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

Only one DBI or SSU can be saved per ECB; for example, it is not possible to nest saved DBI or SSUs.

## Examples

The following example saves the existing DBI and SSU, changes to the BSS, and restores the original DBI and SSU.

```
#include <sysapi.h>
:
:
cebic_save();
cebic_goto_bss();
:
:
cebic_restore();
```

## Related Information

- “cebic\_goto\_bss—Change MDBF Subsystem to BSS” on page 25
- “cebic\_goto\_dbi—Change MDBF Subsystem to DBI” on page 26
- “cebic\_goto\_ssu—Change MDBF Subsystem” on page 27
- “cebic\_save—Save the Current DBI and SSU IDs” on page 29.

---

## cebic\_save—Save the Current DBI and SSU IDs

This saves the ECB's existing DBI and SSU IDs. They can later be restored by calling `cebic_restore`.

### Format

```
#include <sysapi.h>
void      cebic_save(void);
```

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

Only one DBI/SSU can be saved per ECB; for example, it is not possible to nest saved DBI/SSUs.

## Examples

The following example saves the existing DBI and SSU, changes to the BSS, and restores the original DBI and SSU.

```
#include <sysapi.h>
:
:
cebic_save();
cebic_goto_bss();
:
:
cebic_restore();
```

## Related Information

- “`cebic_goto_bss`—Change MDBF Subsystem to BSS” on page 25
- “`cebic_goto_dbi`—Change MDBF Subsystem to DBI” on page 26
- “`cebic_goto_ssu`—Change MDBF Subsystem” on page 27
- “`cebic_restore`—Restore Previously Saved DBI and SSU IDs” on page 28.

## chdir—Change the Working Directory

This function changes the working directory.

### Format

```
#include <unistd.h>
int chdir(const char *pathname);
```

#### **pathname**

The name of the directory that you want to be the new working directory.

This function makes **pathname** your new working directory.

### Normal Return

If successful, the `chdir` function changes the current working directory and returns 0.

### Error Return

If unsuccessful, the `chdir` function does not change the working directory. Instead, it returns `-1`, and sets `errno` to one of the following:

- EACCES**      The process does not have search permission on one of the components of **pathname**.
- ELOOP**      A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of **pathname** is greater than `POSIX_SYMLINK_MAX` (a value defined in the `limits.h` header file).
- ENAMETOOLONG**      **pathname** is longer than `PATH_MAX` characters, or some component of **pathname** is longer than `NAME_MAX` characters. For symbolic links, the length of the **pathname** string substituted for a symbolic link exceeds `PATH_MAX`.
- ENOENT**      **pathname** is an empty string or the specified directory does not exist.
- ENOTDIR**      Some component of **pathname** is not a directory.

### Programming Considerations

None.

### Examples

The following example shows the use of the `chdir` function.

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

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

#### **Output**

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

## Related Information

- “closedir—Close a Directory” on page 46
- “getcwd—Get Path Name of the Working Directory” on page 252
- “mkdir—Make a Directory” on page 326
- “opendir—Open a Directory” on page 384
- “readdir—Read an Entry from a Directory” on page 415
- “rewinddir—Reposition a Directory Stream to the Beginning” on page 433.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## chmod—Change the Mode of a File or Directory

This function changes the mode of a file or directory.

### Format

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

#### pathname

The name of the file or directory whose access mode is to be changed.

#### mode

The **mode** parameter is created with one of the following symbols defined in the `sys/stat.h` header file. Any mode flags that are not specified will be turned off.

##### S\_ISUID

Privilege to set the user ID (UID) to run. When this file is run through the `tpf_fork` function, the effective user ID of the process is set to the owner of the file. The process then has the same authority as the file owner rather than the authority of the actual caller.

##### S\_ISGID

Privilege to set the group ID (GID) to run. When this file is run through the `tpf_fork` function, the effective group ID of the process is set to the group of the file. The process then has the same authority as the file group rather than the authority of the actual caller.

##### S\_IRUSR

Read permission for the file owner.

##### S\_IWUSR

Write permission for the file owner.

##### S\_IXUSR

Search permission (for a directory) for the file owner.

##### S\_IRWXU

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

##### S\_IRGRP

Read permission for the file group.

##### S\_IWGRP

Write permission for the file group.

##### S\_IXGRP

Search permission (for a directory) for the file group.

##### S\_IRWXG

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

##### S\_IROTH

Read permission for users other than the file owner.

##### S\_IWOTH

Write permission for users other than the file owner.

##### S\_IXOTH

Search permission for a directory for users other than the file owner.

**S\_IRWXO**

Read, write, and search permission for users other than the file owner. S\_IRWXO is the bitwise inclusive OR of S\_IROTH, S\_IWOTH, and S\_IXOTH.

This function changes the mode of the file or directory specified in **pathname**.

A process can set mode bits only if the effective user ID of the process is the same as the file's owner or if the process has appropriate privileges (superuser authority). The chmod function automatically clears the S\_ISGID bit in the file's mode bits if all the following conditions are true:

- The calling process does not have appropriate privileges, that is, superuser authority (UID=0).
- The group ID of the file does not match the effective group ID of the calling process.
- One or more of the S\_IXUSR, S\_IXGRP, or S\_IXOTH bits of the file mode are set to 1.
- The file is a regular file.

**TPF deviation from POSIX**

The TPF system does not support the ctime time stamp.

**Normal Return**

If successful, the chmod function returns 0.

**Error Return**

If unsuccessful, chmod returns -1 and sets errno to one of the following:

<b>EACCES</b>	The process does not have search permission on some component of the <b>pathname</b> prefix.
<b>ELOOP</b>	A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of <b>pathname</b> is greater than POSIX_SYMLINK_MAX (a value defined in the limits.h header file).
<b>ENAMETOOLONG</b>	<b>pathname</b> is longer than PATH_MAX characters, or some component of <b>pathname</b> is longer than NAME_MAX characters. For symbolic links, the length of the <b>pathname</b> string substituted for a symbolic link exceeds PATH_MAX.
<b>ENOENT</b>	There is no file named <b>pathname</b> , or the <b>pathname</b> parameter is an empty string.
<b>ENOTDIR</b>	Some component of the <b>pathname</b> prefix is not a directory.
<b>EPERM</b>	The calling process does not have appropriate privileges. The UID of the calling process is neither zero nor the same UID as the file owner.

**Programming Considerations**

None.

## chmod

### Examples

The following example changes the permission from the file owner to the file group.

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

main() {
    char fn[]="/temp.file";
    FILE *stream;
    struct stat info;

    if ((stream = fopen(fn, "w")) == NULL)
        perror("fopen() error");
    else {
        fclose(stream);
        stat(fn, &info);
        printf("original permissions were: %08x\n", info.st_mode);
        if (chmod(fn, S_IRWXU|S_IRWXG) != 0)
            perror("chmod() error");
        else {
            stat(fn, &info);
            printf("after chmod(), permissions are: %08x\n", info.st_mode);
        }
        unlink(fn);
    }
}
```

#### Output

```
original permissions were: 030001b6
after chmod(), permissions are: 030001f8
```

### Related Information

- “chown—Change the Owner or Group of a File or Directory” on page 35
- “fchmod—Change the Mode of a File or Directory by Descriptor” on page 123
- “mkdir—Make a Directory” on page 326
- “open—Open a File” on page 380.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



## chown—Change the Owner or Group of a File or Directory

This function changes the owner or group of a file or directory.

### Format

```
#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group)
```

#### **pathname**

The name of the file whose owner and group are to be changed.

#### **owner**

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

#### **group**

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

A process can change the ownership of a file if the effective user ID of the process is equal to the user ID of the file owner or if the process has appropriate privileges (superuser authority). If the file is a regular file and the process does not have superuser authority, the chown function clears the **S\_ISUID** and **S\_ISGID** bits in the mode of the file.

#### TPF deviation from POSIX

The TPF system does not support the ctime time stamp.

### Normal Return

If successful, chown updates the owner and group for the file and returns 0.

### Error Return

If unsuccessful, chown returns -1 and sets errno to one of the following:

<b>EACCES</b>	The process does not have search permission on some component of the <b>pathname</b> prefix.
<b>EINVAL</b>	<b>owner</b> or <b>group</b> is not a valid user ID (UID) or group ID (GID).
<b>ELOOP</b>	A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of <b>pathname</b> is greater than POSIX_SYMLINK (a value defined in the limits.h header file).
<b>ENAMETOOLONG</b>	<b>pathname</b> is longer than PATH_MAX characters, or some component of <b>pathname</b> is longer than NAME_MAX characters. For symbolic links, the length of the <b>pathname</b> string substituted for a symbolic link exceeds PATH_MAX.
<b>ENOENT</b>	There is no file named <b>pathname</b> or the <b>pathname</b> parameter is an empty string.
<b>ENOTDIR</b>	Some component of the <b>pathname</b> prefix is not a directory.
<b>EPERM</b>	The user ID of the calling process does not match the owner of the file.

## chown

# Programming Considerations

None.

## Examples

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

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

main() {
    char fn[]="temp.file";
    FILE *stream;
    struct stat info;

    if ((stream = fopen(fn, "w")) == NULL)
        perror("fopen() error");
    else {
        fclose(stream);
        stat(fn, &info);
        printf("original owner was %d and group was %d\n", info.st_uid,
            info.st_gid);
        if (chown(fn, 25, 0) != 0)
            perror("chown() error");
        else {
            stat(fn, &info);
            printf("after chown(), owner is %d and group is %d\n",
                info.st_uid, info.st_gid);
        }
        unlink(fn);
    }
}
```

### Output

```
original owner was 0 and group was 0
after chown(), owner is 25 and group is 0
```

## Related Information

- “chmod—Change the Mode of a File or Directory” on page 32
- “fchmod—Change the Mode of a File or Directory by Descriptor” on page 123
- “fchown—Change the Owner and Group by File Descriptor” on page 125
- “fstat—Get Status Information about a File” on page 230
- “lstat—Get Status of a File or Symbolic Link” on page 317
- “stat—Get File Information” on page 513.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## cifrc–Cipher Program Interface

An ECB-controlled program calls the `cifrc` function whenever a portion of a message needs to be enciphered or deciphered. Calling the `cifrc` function causes the IBM-supplied cipher programs BQKCIPH or BQKDES to be executed.

### Format

```
#include <tpfapi.h>
void cifrc(int cipher_program, struct cifrc_arg *c_req);
```

#### **cipher\_program**

Code the defined terms **CIFRC\_AET** to activate IBM-supplied cipher program BQKCIPH, or **CIFRC\_DES** to activate cipher program BQKDES.

#### **c\_req**

Pointer to structure `cifrc_arg`, which contains the request code, a pointer to an 8-byte key, and a pointer to the text to be ciphered. (See the `tpfapi.h` header file.)

### Normal Return

Void. The text pointed to by `ciftext` has been replaced by the enciphered or deciphered text.

### Error Return

Not applicable.

## Programming Considerations

- The data enciphering support is intended to be used to interface with the IBM 3614 Consumer Transaction Facility. See the *3600 Finance Communication System 3614 Programmer's Guide and Reference*.
- When **CIFRC\_AET** is specified, the first 4 bytes of text are processed. When **CIFRC\_DES** is specified, the first 8 bytes of text are processed.
- If the requested cipher program has not been generated in the system, a system error with exit results.

## Examples

The following example enciphers the first 4 bytes of text starting at EBX000, using the 8-byte key at EBW000.

```
#include <tpfapi.h>

/* storage for cifrc_arg */
struct cifrc_arg *reqptr = (struct cifrc_arg *) &(ecbptr()->ebw032);
:
:
reqptr->cifreqst = 0; /* Encipher the Text */
reqptr->cifkey = &(ecbptr()->ebw000); /* pointer to key in EBW000-007 */
reqptr->ciftext = &(ecbptr()->ebx000); /* text to be enciphered */
cifrc(CIFRC_AET, reqptr);
```

## Related Information

None.

## cinfc–Control Program Interface

This function allows ECB-controlled programs to access protected main storage by returning a pointer to an interface point or location at execution time. Additionally, the function provides a convenient way for a real-time program to request an update of the file copy of 1 or more control program keypoint records.

### Format

```
#include <c$cinfc.h>
void *cinfc(enum t_cinfc type, int number, ...);
```

#### type

This parameter must belong to the enumeration type `t_cinfc` specifying 1 of 3 possible values:

##### **CINFC\_ADDRESS**

Specifies that the function is to return an interface location address and allow the calling function to write in protected storage.

##### **CINFC\_WRITE**

Specifies that the function is to return an interface address and allow the calling function to write in protected storage.

##### **CINFC\_READ**

Specifies that the function is to return an interface address and the current protection key is set to that of application core.

##### **CINFC\_KEYPOINT**

Specifies that the function is to initiate CP keypoint update.

#### number

If the type specified is `CINFC_ADDRESS`, `CINFC_WRITE`, or `CINFC_READ`, this parameter is the symbolic label of the interface point. If the type specified is `CINFC_KEYPOINT`, this parameter is the number of keypoints to update.

... One to 5 parameters of keypoints to be updated. Valid entries are `CINFC_KEYB`, `CINFC_KEYD`, `CINFC_KEYE`, `CINFC_KEY1`, or `CINFC_KEY2`.

### Normal Return

Pointer to interface address, or location if requested. For **CINFC\_KEYPOINT**, the return is unknown.

### Error Return

Not applicable.

### Programming Considerations

- `CINFC_KEYPOINT` can only be specified from a program executing on the main I-stream. `CINFC_ADDRESS`, `CINFC_READ`, and `CINFC_WRITE` can be specified from functions executing on any I-stream.
- For `CINFC_KEYPOINT`, keypoint updating will occur at the next opportunity.
- The `CINFC_WRITE` option changes the protection key. Restore the protection key, to the key assigned to working storage, by using the `keyrc` function when write access to the storage is no longer needed. The `CINFC_WRITE` option saves the protection key in effect up to the `cinfc` function call. Restore the original (saved) protection key by using the `keyrc_okey` function.
- If an incorrect parameter is given, a system error with exit occurs.

## Examples

The following example returns the address of a field in protected storage and does not allow the function to modify the field.

```
#include <c$cinf.h>
:
char *field_ptr;
:
field_ptr = cinfc(CINFC_READ, CINFC_CMMINC);
:
```

The following example updates the file copy of 2 control program keypoints.

```
#include <c$cinf.h>
:
cinf(CINFC_KEYPOINT, 2, CINFC_KEY1, CINFC_KEYE);
:
```

## Related Information

- “cinf\_fast—Fast Control Program Interface” on page 40
- “cinf\_fast\_ss—Fast Control Program Interface for Any Active Subsystem” on page 41.

---

## cinf\_fast–Fast Control Program Interface

This function allows ECB-controlled programs to quickly access protected main storage by returning a pointer to an interface point at execution time. This function can only access basic subsystem (BSS) interface points.

### Format

```
#include <c$cinf.h>
void *cinf_fast(int tag);
```

#### tag

This parameter is the symbolic label of the interface point.

### Normal Return

Pointer to interface address.

### Error Return

Not applicable.

### Programming Considerations

- This function leaves the storage accessed as protected.
- This function can only be used to obtain read access to CINFC tags for the BSS subsystem. It should not be used for subsystem-unique areas because it will get the pointer to the BSS area instead of the subsystem area.

### Examples

The following example returns the address of a field and does not change the protect key.

```
#include <c$cinf.h>
...
char *field_ptr;
...
field_ptr = (char *)cinf_fast(CINFC_CMMINC);
...
```

### Related Information

- “cinf–Control Program Interface” on page 38
- “cinf\_fast\_ss–Fast Control Program Interface for Any Active Subsystem” on page 41.

## cinf\_fast\_ss—Fast Control Program Interface for Any Active Subsystem

This function allows ECB-controlled programs to quickly access protected main storage by returning a pointer to an interface point at execution time. This function can access interface points for any active subsystem.

### Format

```
#include <c$cinf.h>
void      *cinf_fast_ss(int tag, int ssindex);
```

#### tag

This parameter is the symbolic label of the interface point.

#### ssindex

This parameter is the subsystem index.

**Note:** Only bits 24–31 of the **ssindex** are used. If the subsystem is not valid, results are indeterminate.

### Normal Return

Pointer to interface address.

### Error Return

Not applicable.

### Programming Considerations

- This function leaves the storage accessed as protected.
- This function can only be used to obtain read access to CINFC tags for any active subsystem.

### Examples

The following example returns the address of a field in the subsystem where the ECB is running and does not change the protect key.

```
#include <c$cinf.h>
#include <c$eb0eb.h>
:
:
char *field_ptr;
:
:
field_ptr = (char *)cinf_fast_ss(CINFC_CMMOPS, ecbptr()->celdbi);
:
:
```

### Related Information

- “cinf—Control Program Interface” on page 38
- “cinf\_fast—Fast Control Program Interface” on page 40.

---

## clearerr–Reset Error and End-of-File

This function resets the error indicator and end-of-file (EOF) indicator for the stream that `stream` points to. Generally, the indicators for a stream remain set until your program calls the `clearerr` or `rewind` function.

### Format

```
#include <stdio.h>
void clearerr (FILE *stream);
```

#### **stream**

The open stream whose error and end-of-file (EOF) indicator are to be reset.

### Normal Return

Returns no value.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example reads a data stream and then checks that a read error has not occurred.

```
#include <stdio.h>

int main(void)
{
    char string[100];
    FILE *stream;
    int eofvalue;

    stream = fopen("myfile.dat", "r");

    /* scan an input stream until an end-of-file character is read */
    while (!feof(stream))
        fscanf(stream,"%s",&string[0]);

    /* print EOF value: will be nonzero */
    eofvalue=feof(stream);
    printf("feof value=%i\n",eofvalue);

    /* print EOF value-after clearerr, will be equal to zero */
    clearerr(stream);
    eofvalue=feof(stream);
    printf("feof value=%i\n",eofvalue);
}
```

### Related Information

- “feof–Test End-of-File Indicator” on page 141
- “ferror–Test for Read/Write Errors” on page 143
- “fseek–Change File Position” on page 226
- “rewind–Set File Position to Beginning of File” on page 431.



See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## close—Close a File

This function closes a file.

### Note

This description applies only to files. See *TPF Transmission Control Protocol/Internet Protocol* for more information about the `close` function for sockets.

## Format

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

### fildes

The file descriptor of the open file that is to be closed.

This function closes file descriptor **fildes**. This frees the file descriptor to be returned by future open calls and other calls that create file descriptors.

When the last open file descriptor for a file is closed, the file itself is closed.

The `close` function unlocks (removes) all outstanding record locks that the process has on the associated file.

### TPF deviation from POSIX

If the file was opened with write access or with read and write access, `close` updates the `st_mtime` field with the current time.

## Normal Return

If successful, the `close` function returns 0.

## Error Return

If unsuccessful, the `close` function returns `-1` and sets `errno` to the following:

**EBADF**            **fildes** is not a valid open file descriptor.

## Programming Considerations

None.

## Examples

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

main() {
    int fd;
    char out[20]="Test string";
    if ((fd = creat("my.file")) < 0)
        perror("creat error");
    else {
        if (write(fd, out, strlen(out)+1) == -1) {
            perror("write() error");
        }
    }
}
```

```
        if (fd = 0) perror("close() error");  
        close(fd);  
    }  
}
```

## Related Information

- “abort—Terminate Program Abnormally” on page 7
- “creat—Create a New File or Rewrite an Existing File” on page 54
- “dup—Duplicate an Open File Descriptor” on page 97
- “exit—Exit an ECB” on page 115
- “fclose—Close File Stream” on page 127
- “fcntl—Control Open File Descriptors” on page 129
- “open—Open a File” on page 380
- “unlink—Remove a Directory Entry” on page 668.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## closedir—Close a Directory

This function closes a directory.

### Format

```
#include <dirent.h>
int closedir(DIR *dir);
```

#### **dir**

The handle returned by `opendir` for the open directory that is to be closed.

This function closes the directory indicated by **dir** and frees the buffer that the `readdir` function uses when reading the directory stream.

### Normal Return

If successful, the `closedir` function returns 0.

### Error Return

If unsuccessful, the `closedir` function returns `-1` and sets `errno` to the following:

**EBADF**            **dir** does not refer to an open directory stream.

### Programming Considerations

None.

### Examples

The following example closes a directory.

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

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

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

#### **Output**

```
directory entry 001: .
directory entry 002: ..
directory entry 003: dev
directory entry 004: etc
directory entry 005: lib
directory entry 006: tmp
directory entry 007: u
directory entry 008: usr
```

## **Related Information**

- “[opendir](#)—Open a Directory” on page 384
- “[readdir](#)—Read an Entry from a Directory” on page 415
- “[rewinddir](#)—Reposition a Directory Stream to the Beginning” on page 433.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## closelog—Close the System Control Log

This function closes the log file. Use this function with the `openlog` function and the `syslog` function to use the logging facilities provided with the syslog daemon.

See *TPF Transmission Control Protocol/Internet Protocol* for more information about the syslog daemon.

### Format

```
#include <syslog.h>
void    closelog(void);
```

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- This function closes any open file descriptors that were allocated by previous calls to the `openlog` or `syslog` function.
- If you do not code the `closelog` function, the log file is closed automatically when the entry control block (ECB) exits.
- This function is implemented in dynamic link library (DLL) CTXO. You must use the definition side-deck for DLL CTXO to link-edit an application that uses this function.

### Examples

See “syslog—Send a Message to the Control Log” on page 523 for an example of using the `closelog` function.

### Related Information

- “openlog—Open the System Control Log” on page 386
- “syslog—Send a Message to the Control Log” on page 523.

## corhc—Define and Hold a Resource

This function requests exclusive control over a resource. For example, to update a storage area such as a global record that is shared across processors in a multiprocessing environment, obtaining a pointer to the global and requesting corhc against it forces serialization on the global record during update.

### Format

```
#include <tpfapi.h>
void corhc(void *resource);
```

#### resource

A pointer to type void representing the resource over which exclusive control is being requested.

### Normal Return

Void. Control is not returned to the issuing ECB until exclusive control over the resource is obtained.

### Error Return

Not applicable.

## Programming Considerations

When finished with the resource, the ECB must call coruc to free the resource. The pointer value (not the referenced value) passed to coruc must be the same as the pointer value passed to corhc. The pointer value is not required to reference a storage location.

## Examples

The following example obtains a hold on global record \_q05met based on the address of the record.

```
#include <tpfapi.h>
#include <tpfglbl.h>
#include <c$globz.h>
struct msgexp *q05met;
:
q05met = *(struct msgexp **) glob(_q05met);
corhc(q05met);
```

The following example obtains a lock on the resource name ABCD.

```
#include <tpfapi.h>
#define RESOURCE_KEY ((void *) 0xC1C2C3C4)
:
corhc(RESOURCE_KEY);
```

## Related Information

- “coruc—Unhold a Resource” on page 50
- “glob—Address TPF Global Field or Record” on page 277.

## coruc—Unhold a Resource

This function releases exclusive control over a resource.

### Format

```
#include <tpfapi.h>
void coruc(void *resource);
```

#### resource

A pointer to type void representing the resource over which exclusive control is to be relinquished.

### Normal Return

Void. Exclusive control over the resource has been relinquished.

### Error Return

Not applicable.

## Programming Considerations

Attempting to release a resource that the ECB is not holding results in a system error with exit. The pointer value (not the referenced value) passed to coruc must be the same as the pointer value passed to corhc. The pointer value is not required to reference a storage location.

## Examples

The following example obtains a hold on the global record \_q05met based on the address of the record, and then relinquishes the hold.

```
#include <tpfapi.h>
#include <tpfglbl.h>
#include <c$globz.h>
struct msgexp *q05met;
:
:
q05met = *(struct msgexp **) glob(_q05met);
corhc(q05met);
:
:
coruc(q05met);
modex(MODEC_24);
```

The following example obtains a lock on resource name ABCD, and then relinquishes the lock.

```
#include <tpfapi.h>
#define RESOURCE_KEY ((void *) 0xC1C2C3C4)
:
:
corhc(RESOURCE_KEY);
:
:
coruc(RESOURCE_KEY);
```

## Related Information

- “corhc—Define and Hold a Resource” on page 49
- “glob—Address TPF Global Field or Record” on page 277.



## cratc—Search CRAS Status Table

This function searches the CRAS status table.

### Format

```
#include <tpfapi.h>
long int cratc(unsigned long type, struct cratc_iparms *cratc_input,
              struct cratc_slot **addr_crat_slot_ptr);
```

#### type

One of the following values that identify the type of search being requested:

##### CRATC\_VERIFY

Verifies that a terminal address is in the CRAS status table.

##### CRATC\_LOCATE

Obtains the LNIATA/CPU ID and CRAS table slot address of a terminal. The slot address is returned for either 1) a functional support console (FSC) when given a routing code indicator and a CPU ID or 2) an alternate CRAS slot (printer or terminal index).

##### CRATC\_FSC

Obtains the LNIATA/CPU ID and CRAS table slot address of a terminal. The CRAS slot address is returned when given an FSC routing index located at the beginning of the CRAS table.

#### cratc\_input

A pointer to the cratc\_iparms structure.

- If CRATC\_VERIFY is the specified **type**, the cratc\_iparms structure must contain the LNIATA and, optionally, CPU ID of the terminal. If the CPU ID is omitted, you must zero out the CPU ID field in the input structure and the CPU ID will default to that of the ECB (CE1CPD).
- If CRATC\_LOCATE is the specified **type**, the cratc\_iparms structure must contain one of the following:
  - The FSC slot address and, optionally, the CPU ID. For this, the following routing codes can be specified in the cratc\_iparms structure:
    - WTOPC\_AUDT
    - WTOPC\_COMM
    - WTOPC\_DASD
    - WTOPC\_PRC
    - WTOPC\_RDBS
    - WTOPC\_RO
    - WTOPC\_TAPE
  - An alternate CRAS slot index (terminal or printer index).

If the CPU ID is omitted, you must zero out the CPU ID field in the input structure and the CPU ID will default to that of the ECB (CE1CPD).

- If CRATC\_FSC is the specified **type**, the cratc\_iparms structure must contain the CRAS table index.

#### addr\_crat\_slot\_ptr

A pointer to a 4-byte field in which the CRAS table slot address will be placed on a normal return from the cratc function call.

### Normal Return

A nonzero long integer. When this value is converted to hexadecimal format, it is equivalent to the concatenation of the 3 bytes of the LNIATA number (also in hex) with the last byte of the EBCDIC CPU ID (converted to hex) of the located CRAS

## cratc

terminal. The return code integer can also be equated to the integer value in the `cratc_oparms` structure. Because this structure is a union, the integer can then be addressed in character format.

The CRAS table slot address will be stored in the location pointed to by the `addr_crat_slot_ptr` parameter.

## Error Return

A value of 0. This indicates that the requested information could not be located in the CRAS table.

## Programming Considerations

None.

## Examples

The following example does a `CRATC_VERIFY` on LNIATA 010000 for CPU B.

```
#include <tpfeq.h>
#include <tpfio.h>
#include <tpfapi.h>
#include <string.h>
#include <stdio.h>
:
struct cratc_slot * crat_slot_ptr;
struct cratc_iparms cratc_input;
struct cratc_oparms cratc_output;
unsigned long int rc;
unsigned char lniata[3] = {01,00,00};
unsigned char cpuid = 'B';
:
memset(&cratc_input,0x00,sizeof(struct cratc_iparms));
memcpy(cratc_input.cr_in.cratc_i_lniata,lniata,3);
memcpy(&cratc_input.cratc_i_cpuid,&cpuid,1);
rc = cratc(CRATC_VERIFY, &cratc_input, &crat_slot_ptr);
if (rc)
{
    cratc_output.cr_out.cratc_o_return = rc;
}
:
exit(0);
}
```

The following example does a `CRATC_LOCATE` for the PRC routing code for CPU B.

```
#include <tpfeq.h>
#include <tpfio.h>
#include <tpfapi.h>
#include <string.h>
#include <stdio.h>
:
struct cratc_slot * crat_slot_ptr;
struct cratc_iparms cratc_input;
struct cratc_oparms cratc_output;
unsigned long int rc;
unsigned char cpuid = 'B';
short int routecd = WTOPC_PRC;
:
memset(&cratc_input,0x00,sizeof(struct cratc_iparms));
memcpy(&cratc_input.cr_in.cratc_i_rtcd[1],&routecd,2);
memcpy(&cratc_input.cratc_i_cpuid,&cpuid,1);
rc = cratc(CRATC_LOCATE, &cratc_input, &crat_slot_ptr);
```

```

if (rc)
{
    cratc_output.cr_out.cratc_o_return = rc;
}
:
:
exit(0);
}

```

The following example does a CRATC\_FSC with the CRAS table index 3.

```

#include <tpfeq.h>
#include <tpfio.h>
#include <tpfapi.h>
#include <string.h>
#include <stdio.h>
:
:
struct cratc_slot * crat_slot_ptr;
struct cratc_iparms cratc_input;
struct cratc_oparms cratc_output;
unsigned long int rc;
unsigned char index = 3;
:
:
memset(&cratc_input,0x00,sizeof(struct cratc_iparms));
memcpy(&cratc_input.cr_in.cratc_i_rtcd[0],&index,1);
rc = cratc(CRATC_FSC, &cratc_input, &crat_slot_ptr);
if (rc)
{
    cratc_output.cr_out.cratc_o_return = rc;
}
:
:
exit(0);
}

```

## Related Information

“wgtac—Locate Terminal Entry” on page 691.

## creat—Create a New File or Rewrite an Existing File

This function creates a new file or rewrites an existing one.

### Format

```
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

#### pathname

The name of the file to be created.

#### mode

The **mode** parameter specifies the file permission bits to be used in creating the file. The following is a list of symbols that you can use for a mode.

##### S\_ISUID

Privilege to set the user ID (UID) to run. When this file is run through the `tpf_fork` function, the effective user ID of the process is set to the owner of the file. The process then has the same authority as the file owner rather than the authority of the actual caller.

##### S\_ISGID

Privilege to set the group ID (GID) to run. When this file is run through the `tpf_fork` function, the effective group ID of the process is set to the group of the file. The process then has the same authority as the file group rather than the authority of the actual caller. If **pathname** is a regular file and the calling process is not running as superuser (its effective UID is not 0), the `S_ISUID` and `S_ISGID` flags for the file are cleared to zeros.

##### S\_IRUSR

Read permission for the file owner.

##### S\_IWUSR

Write permission for the file owner.

##### S\_IXUSR

Search permission (for a directory) for the file owner.

##### S\_IRWXU

Read, write, and search permission for the file owner; `S_IRWXU` is the bitwise inclusive OR of `S_IRUSR`, `S_IWUSR`, and `S_IXUSR`.

##### S\_IRGRP

Read permission for the file group.

##### S\_IWGRP

Write permission for the file group.

##### S\_IXGRP

Search permission (for a directory) for the file group.

##### S\_IRWXG

Read, write, and search permission for the file group. `S_IRWXG` is the bitwise inclusive **OR** of `S_IRGRP`, `S_IWGRP`, and `S_IXGRP`.

##### S\_IROTH

Read permission for users other than the file owner or group.

##### S\_IWOTH

Write permission for users other than the file owner or group.

**S\_IXOTH**

Search permission for a directory for users other than the file owner or group.

**S\_IRWXO**

Read, write, and search permission for users other than the file owner or group. S\_IRWXO is the bitwise inclusive OR of S\_IROTH, S\_IWOTH, and S\_IXOTH.

The function call: `creat(pathname,mode)` is equivalent to the call:

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

Therefore, the file named by **pathname** is created unless it already exists. The file is then opened for writing only and is truncated to zero length. See “open–Open a File” on page 380 for more information.

## Normal Return

If the `creat` function is successful, it returns a file descriptor for the open file.

## Error Return

If unsuccessful, the `creat` function returns `-1` and sets `errno` to one of the following:

**EACCES**

- The process did not have search permission on a component in **pathname**.
- The file exists, but the process did not have appropriate permissions to open the file in the way specified by the flags.
- The file does not exist and the process does not have write permission on the directory where the file is to be created.

**EISDIR**

**pathname** is a directory and **options** specifies write or read/write access.

**ELOOP**

A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of **pathname** is greater than `POSIX_SYMLLOOP`.

**EMFILE**

The process has reached the maximum number of file descriptors it can have open.

**ENAMETOOLONG**

**pathname** is longer than `PATH_MAX` characters, or some component of **pathname** is longer than `NAME_MAX` characters.

For symbolic links, the length of the **pathname** string substituted for a symbolic link exceeds `PATH_MAX`.

**ENFILE**

The system has reached the maximum number of file descriptors it can have open.

**ENOENT**

Either the path prefix does not exist or the **pathname** parameter is an empty string.

**ENOSPC**

The directory or file system that intended to hold a new file does not have enough space.

**ENOTDIR**

A component of **pathname** is not a directory.

## creat

### Programming Considerations

- The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.
- When a new regular file is created, if the TPF\_REGFILE\_RECORD\_ID environment variable is set to a 2-character string, its value is used as the record ID for all pool file records that are allocated to store the contents of the file.

### Examples

The following example creates a file.

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

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

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

### Related Information

- “close—Close a File” on page 44
- “open—Open a File” on page 380
- “unlink—Remove a Directory Entry” on page 668.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## credc, \_\_CREDC—Create a Deferred Entry

This macro creates an independent ECB for deferred processing.

A variable number of bytes may be passed to the created ECB's work area starting at EBW000. The TPF system moves the parameters into an interim block of available storage and adds this block to the deferred processing CPU loop list. Operational program zero (OPZERO) initializes an ECB with the parameters in the work area, releases the interim block, and activates the specified program.

### Format

```
#include <tpfapi.h>
void credc(int length, const void *parm, void (*segname)());
void __CREDC(int length, const void *parm, const char *segname);
```

#### length

An integer containing the number of bytes to be passed to the created ECB. A value of zero indicates that no parameters will be passed. A maximum of 104 bytes may be passed.

#### parm

A pointer of type void to the parameters to be passed.

#### segname

For credc, a pointer to the external function to be called. For \_\_CREDC, a pointer to the name of the segment to be called. *segname* must map to an assembler segment name, a TARGET(TPF) segment name (or transfer vector), or a TPF ISO-C dynamic load module (DLM) name.

#### Migration to ISO-C consideration

The TARGET(TPF) credc function can handle segment names that are #pragma mapped. The ISO-C credc function only handles #pragma mapped names if the first 4 characters are the same as the program name.

#### TARGET(TPF) restriction

Only calls to certain TPF API functions permit the use of function pointers. Pointers to functions are not supported in TARGET(TPF); their use normally results in a compiler error.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- No linkage is provided between the created ECB and the active ECB calling this macro.
- The ECB calling credc may be forced into a wait if there is insufficient storage available to buffer the parameters. When adequate working storage becomes available, the macro is executed and control is returned.
- ECBs are started on the I-stream where the create macro was issued.

## **credc, \_\_CREDC**

- Use of this macro should be limited to prevent a depletion of storage.
- A system error with exit occurs if no block is held on the specified level, or if more than 104 bytes of parameters are passed.

## **Examples**

The following example creates a deferred entry for program COT0, passing string VPH as input data to the program.

```
#include <tpfapi.h>
void COT0();
char *parmstring = "VPH";
:
credc(strlen(parmstring),parmstring,COT0);
```

## **Related Information**

- “creec, \_\_CREEC—Create a New ECB with an Attached Block” on page 59
- “cremc, \_\_CREMC—Create an Immediate Entry” on page 62
- “cretc, \_\_CRETCL—Create a Time-Initiated Entry” on page 64
- “cretc\_level, \_\_CRETCL—Create a Time-Initiated Entry” on page 66
- “crexc, \_\_CREXC—Create a Low-Priority Deferred Entry” on page 69.



## creec, \_\_CREEC—Create a New ECB with an Attached Block

This macro creates an independent ECB for immediate or deferred processing.

A core block is placed on data level zero of the newly created ECB and a variable number of bytes is passed to the created ECB's work area starting at EBW000. The core block to be passed is the one specified by the **level** or **decb** parameter. The TPF system moves the parameters from the area specified by the **parm** parameter into an interim block of storage. If the **priority** parameter is **CREEC\_IMMEDIATE**, the block is placed on the ready list. If **CREEC\_DEFERRED**, it is placed on the deferred list. Operational program zero (OPZERO) initializes an ECB with the parameters in the work area, releases the interim block, places the passed block (if any) on data level zero, and activates the specified program.

### Format

```
#include <tpfapi.h>
void creec(int length, const void *parm, void (*segname)(),
           enum t_lvl level, int priority);
void __CREEC(int length, const void *parm, const char *segname,
             enum t_lvl level, int priority);
```

or

```
#include <tpfapi.h>
void creec(int length, const void *parm, void (*segname)(),
           TPF_DECB *decb, int priority);
void __CREEC(int length, const void *parm, const char *segname,
             TPF_DECB *decb, int priority);
```

#### length

An integer containing the number of bytes to be passed to the created ECB. A value of zero indicates that no parameters will be passed. A maximum of 104 bytes may be passed.

#### parm

A pointer of type void to the parameters to be passed.

#### segname

For **creec**, a pointer to the external function to be called. For **\_\_CREEC**, a pointer to the name of the segment to be called. *segname* must map to an assembler segment name, a TARGET(TPF) segment name (or transfer vector), or a TPF ISO-C dynamic load module (DLM) name.

#### Migration to ISO-C consideration

The TARGET(TPF) **creec** function can handle segment names that are `#pragma mapped`. The ISO-C **creec** function only handles `#pragma mapped` names if the first 4 characters are the same as the program name.

#### TARGET(TPF) restriction

Only calls to certain TPF API functions permit the use of function pointers. Pointers to functions are not supported in TARGET(TPF); their use normally results in a compiler error.

#### level

One of 16 possible values representing a valid data level from the enumeration

**creec, \_\_CREEC**

type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this CBRW level is the block to be passed to the newly created ECB.

**decb**

A pointer to a data event control block (DECB). The working storage block on this DECB is the block to be passed to the newly created ECB.

**priority**

Use defined terms **CREEC\_DEFERRED** to indicate placement on the deferred list and **CREEC\_IMMEDIATE** to indicate placement on the ready list.

## Normal Return

Void.

## Error Return

Not applicable.

## Programming Considerations

- On return, the working storage block on the referenced ECB data level or DECB is no longer available to the calling program.
- No linkage is provided between the created ECB and the active ECB calling this macro.
- The ECB calling `creec` may be forced into a wait state if there is insufficient storage available to buffer the parameters. When adequate working storage becomes available, the macro is executed and control is returned.
- ECBs are started on the I-stream where the create macro was issued.
- The use of this macro should be limited to prevent a depletion of storage.
- A system error with exit occurs if no block is held on the specified ECB data level or DECB, or if more than 104 bytes are passed.
- If you use this macro to create an ECB that will enter a dynamic load module (DLM) with an entry point defined by the main function, the TPF system assumes that any core block attached to data level 0 (D0) contains a command string that will be parsed into `argc` and `argv` parameters for the main function. See *TPF Application Programming* for more information about the main function.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example creates an ECB dispatched from the ready list for program OMA0, passing string 755/15AUG and the working storage block on level D0 as input to the program.

```
#include <tpfapi.h>
void OMA0();
char *parmstring = "755/15AUG";
:
creec(strlen(parmstring),parmstring,OMA0,D0,CREEC_IMMEDIATE);
```

The following example creates an ECB dispatched from the ready list for program OMA0, passing string 755/15AUG and the working storage block on the DECB pointed to by **decb\_ptr** as input to the program.

```
#include <tpfapi.h>
void OMA0();
char    *parmstring = "755/15AUG";
TPF_DECB *decb_ptr;
DECBC_RC rc;

decb_ptr = tpf_decb_create( NULL, &rc );
:
creec(strlen(parmstring),parmstring,OMA0,decb_ptr,CREEC_IMMEDIATE);
```

## Related Information

- “credc, \_\_CREDC—Create a Deferred Entry” on page 57
- “cremc, \_\_CREMC—Create an Immediate Entry” on page 62
- “cretc, \_\_CRETCL—Create a Time-Initiated Entry” on page 64
- “cretc\_level, \_\_CRETCL—Create a Time-Initiated Entry” on page 66.

For more information about DECBs, see *TPF Application Programming*.

## cremc, \_\_CREMC—Create an Immediate Entry

This macro creates an independent ECB for immediate processing.

A variable number of bytes may be passed to the created ECB's work area starting at EBW000. The TPF system moves the parameters into an interim block of available storage and adds this block to the ready list. Operational program zero (OPZERO) initializes an ECB with the parameters in the work area, releases the interim block, and activates the specified program.

### Format

```
#include <tpfapi.h>
void      cremc(int length, const void *parm, void (*segname)());
void      __CREMC(int length, const void *parm, const char *segname);
```

#### length

An integer containing the number of bytes to be passed to the created ECB. A value of zero indicates that no parameters will be passed. A maximum of 104 bytes may be passed.

#### parm

A pointer of type void to the parameters to be passed.

#### segname

For cremc, a pointer to the external function being called. For \_\_CREMC, a pointer to the name of the segment to be called. **segname** must map to an assembler segment name, a TARGET(TPF) segment name (or transfer vector), or a TPF ISO-C DLM name.

#### Migration to ISO-C consideration

The TARGET(TPF) cremc function can handle segment names that are #pragma mapped. The ISO-C cremc function only handles #pragma mapped names if the first 4 characters are the same as the program name.

#### TARGET(TPF) restriction

Only calls to certain TPF API functions permit the use of function pointers. Pointers to functions are not supported in TARGET(TPF); their use normally results in a compiler error.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- No linkage is provided between the created ECB and the active ECB calling this macro.
- The ECB calling cremc may be forced into a wait if there is insufficient storage available to buffer the parameters. When adequate working storage becomes available, the macro is executed and control is returned.
- ECBs are started on the I-stream where the create macro was issued.

- The use of this function should be limited to prevent a depletion of storage.

## Examples

The following example creates an ECB dispatched from the ready list for program COT0, passing the string VPH as input to the program.

```
#include <tpfapi.h>
void COT0();
char *parmstring = "VPH";
:
cremc(strlen(parmstring),parmstring,COT0);
```

## Related Information

- “credc, \_\_CREDC—Create a Deferred Entry” on page 57
- “creec, \_\_CREEC—Create a New ECB with an Attached Block” on page 59
- “cretc, \_\_CRETCL—Create a Time-Initiated Entry” on page 64
- “cretc\_level, \_\_CRETCL—Create a Time-Initiated Entry” on page 66
- “crexc, \_\_CREXC—Create a Low-Priority Deferred Entry” on page 69.

## crtc, \_\_CRETC—Create a Time-Initiated Entry

This macro creates an independent ECB after a specified amount of time has elapsed. A 4-byte parameter is placed in EBW000–EBW003 of the newly created ECB.

### Format

```
#include <tpfapi.h>
void crtc(int flags, void (*segname)(), int units,
          const void *action);
void __CRETC(int flags, const char *segname, int units,
             const void *action);
```

#### flags

Logical or (|) of the following bit flags that are defined in tpfapi.h:

#### CRETC\_SECONDS or CRETC\_MINUTES

Use CRETC\_SECONDS or CRETC\_MINUTES to indicate whether the **units** parameter is expressed in seconds or minutes. Code only one of these options.

#### CRETC\_1052

This optional parameter permits the program specified by **seg\_name** to be activated at the time requested, even when the system is in 1052 state. If CRETC\_1052 is not specified, the program cannot be activated until the system is cycled above 1052 state.

#### segname

For crtc, a pointer to the external function to be called. For \_\_CRETC, a pointer to the name of the segment to be called. **segname** must map to an assembler segment name, a TARGET(TPF) segment name (or transfer vector), or a TPF ISO-C DLM name.

#### Migration to ISO-C consideration

The TARGET(TPF) crtc function can handle segment names that are #pragma mapped. The ISO-C crtc function only handles #pragma mapped names if the first 4 characters are the same as the program name.

#### TARGET(TPF) restriction

Only calls to certain TPF API functions permit the use of function pointers. Pointers to functions are not supported in TARGET(TPF); their use will normally result in a compiler error.

#### units

An integer value defining the time increment in minutes or seconds that will elapse before activating the ECB.

**Note:** When you specify the time increment in minutes, an ECB is created at a full minute boundary, which can be less than the interval specified with the **unit** parameter. For example, assume you specified an interval of 1 minute. If the current time is 10:35:55, the new ECB can be created at 10:36:00.

#### action

Treated as a pointer to type void, the first 4 bytes of which are passed to the created ECB as an action word in EBW000–003.

## Normal Return

Void.

## Error Return

Not applicable.

## Programming Considerations

- No linkage is provided between the created ECB and the active ECB calling this macro.
- ECBs are started on the I-stream where the create macro was issued.
- The use of this macro should be limited to prevent a depletion of storage.

## Examples

The following example creates a new ECB for program QZZ0 after 5 seconds have elapsed. The 4-byte character string INIT is placed in EBW000-003 of the new ECB.

```
#include <tpfapi.h>
void QZZ0();
:
cretc(CRETC_SECONDS,QZZ0,5,"INIT");
```

## Related Information

- “credc, \_\_CREDC—Create a Deferred Entry” on page 57
- “creec, \_\_CREEC—Create a New ECB with an Attached Block” on page 59
- “cremc, \_\_CREMC—Create an Immediate Entry” on page 62
- “cretc\_level, \_\_CRETCL—Create a Time-Initiated Entry” on page 66
- “crexc, \_\_CREXC—Create a Low-Priority Deferred Entry” on page 69.

## cretc\_level, \_\_CRETCL—Create a Time-Initiated Entry

This macro creates an independent entry control block (ECB) after a specified amount of time has elapsed. A 4-byte parameter is placed in EBW000–EBW003 of the newly created ECB. A working storage block from an ECB data level or data event control block (DECB) is passed to the created ECB.

### Format

```
#include <tpfapi.h>
void      cretc_level(int flags, void (*segname)(), int units,
                    const void *action, enum t_lvl level);
void      __CRETCL(int flags, const char *segname, int units,
                    const void *action, enum t_lvl level);
```

or

```
#include <tpfapi.h>
void      cretc_level(int flags, void (*segname)(), int units,
                    const void *action, TPF_DECB *decb);
void      __CRETCL(int flags, const char *segname, int units,
                    const void *action, TPF_DECB *decb);
```

#### flags

Logical or (|) of the following bit flags that are defined in tpfapi.h:

#### CRETCL\_SECONDS or CRETCL\_MINUTES

Use CRETCL\_SECONDS or CRETCL\_MINUTES to indicate whether the **units** parameter is expressed in seconds or minutes. Code only one of these options.

#### CRETCL\_1052

This optional parameter permits the program specified by **seg\_name** to be activated at the time requested, even when the system is in 1052 state. If CRETCL\_1052 is not specified, the program cannot be activated until the system is cycled above 1052 state.

#### segname

For cretc\_level, a pointer to the external function to be called. For \_\_CRETCL, a pointer to the name of the segment to be called. *segname* must map to an assembler segment name, a TARGET(TPF) segment name (or transfer vector), or a TPF ISO-C dynamic link module (DLM) name.

#### Migration to ISO-C consideration

The TARGET(TPF) cretc\_level function can handle segment names that are #pragma mapped. The ISO-C cretc\_level function only handles #pragma mapped names if the first 4 characters are the same as the program name.

#### TARGET(TPF) restriction

Only calls to certain TPF API functions permit the use of function pointers. Pointers to functions are not supported in TARGET(TPF); their use normally results in a compiler error.

#### units

An integer value defining the time increment in minutes or seconds that will elapse before activating the ECB.



**Note:** When you specify the time increment in minutes, an ECB is created at a full minute boundary, which can be less than the interval specified with the **units** parameter. For example, assume you specified an interval of 1 minute. If the current time is 10:35:55, the new ECB can be created at 10:36:00.

**action**

Treated as a pointer to type `void`, the first 4 bytes of which are passed to the created ECB as an action word in EBW000–003.

**level**

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this core block reference word (CBRW) level is the block to be passed to the newly created ECB on data level 0.

**decb**

A pointer to a DECB. The working storage block on this DECB is the block to be passed to the newly created ECB on data level 0.

## Normal Return

Void.

## Error Return

Not applicable.

## Programming Considerations

- No linkage is provided between the created ECB and the active ECB calling this macro.
- ECBs are started on the I-stream where the create macro was issued.
- The use of this macro should be limited to prevent a depletion of storage.
- `cretc_level` in `TARGET(TPF)` is supported in NORM state only.
- For `TARGET(TPF)` the STATE parameter of the assembler CRETC macro is not supported.
- If you use this macro to create an ECB that will enter a dynamic load module (DLM) with an entry point defined by the main function, the TPF system assumes that any core block attached to data level 0 (D0) contains a command string that will be parsed into `argc` and `argv` parameters for the main function. See *TPF Application Programming* for more information about the main function.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example creates a new ECB for program QZZ0 after 5 seconds have elapsed. The 4-byte character string, INIT, is placed in EBW000–003 of the new ECB, and data level D2 refers to a core block to be passed to the ECB when it is dispatched.

## cretc\_level, \_\_CRETCL

```
#include <tpfapi.h>
void QZZ0();
:
cretc_level(CRETC_SECONDS,QZZ0,5,"INIT", D2);
```

The following example creates a new ECB for program QZZ0 after 5 seconds have elapsed. The 4-byte character string, INIT, is placed in EBW000–003 of the new ECB, and the core block on the DECB pointed to by **decb\_ptr** is passed to the ECB when it is dispatched.

```
#include <tpfapi.h>
void QZZ0();

TPF_DECB *decb_ptr;
DECBC_RC rc;

decb_ptr = tpf_decb_create( NULL, &rc );
:
cretc_level(CRETC_SECONDS,QZZ0,5,"INIT",decb_ptr);
```

## Related Information

- “credc, \_\_CREDC—Create a Deferred Entry” on page 57
- “creec, \_\_CREEC—Create a New ECB with an Attached Block” on page 59
- “cremc, \_\_CREMC—Create an Immediate Entry” on page 62
- “cretc, \_\_CRETCL—Create a Time-Initiated Entry” on page 64.

For more information about DECBs, see *TPF Application Programming*.

## crexc, \_\_CREXC—Create a Low-Priority Deferred Entry

This macro creates a low-priority independent ECB for deferred processing.

A variable number of bytes may be passed to the created ECB's work area starting at EBW000. The TPF system moves the parameters into an interim block of available storage and adds this block to the deferred list. Operational program zero (OPZERO) initializes an ECB with the parameters in the work area, releases the interim block, and activates the specified program.

### Format

```
#include <tpfapi.h>
void crexc(int length, const void *parm, void (*segname)());
void __CREXC(int length, const void *parm, const char *segname);
```

#### length

An integer containing the number of bytes to be passed to the created ECB. A value of zero indicates that no parameters are to be passed. A maximum of 104 bytes may be passed.

#### parm

A pointer of type void to the parameters to be passed.

#### segname

For crexc, a pointer to the external function to be called. For \_\_CREXC, a pointer to the name of the segment to be called. *segname* must map to an assembler segment name, a TARGET(TPF) segment name (or transfer vector), or a TPF ISO-C dynamic link module (DLM) name.

#### Migration to ISO-C consideration

The TARGET(TPF) crexc function can handle segment names that are #pragma mapped. The ISO-C crexc function only handles #pragma mapped names if the first 4 characters are the same as the program name.

#### TARGET(TPF) restriction

Only calls to certain TPF API functions permit the use of function pointers. Pointers to functions are not supported in TARGET(TPF); their use normally results in a compiler error.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- This macro works similarly to credc except that crexc requires a greater number of available core blocks before the entry is created. The crexc macro should be used by programs that create many entries, such as utilities, in order to limit the use of working storage and creation of entries to minimize the impact on more time-dependent entries.

## crexc, \_\_CREXC

- No linkage is provided between the created ECB and the active ECB calling this macro.
- The ECB calling crexc may be forced into a wait if there is insufficient storage available to buffer the parameters. When adequate working storage becomes available, the macro is executed and control is returned.
- ECBs are started on the I-stream where the create macro was issued.
- The use of this macro should be limited to prevent a depletion of storage.
- A system error with exit occurs if no block is held on the specified level, or if more than 104 bytes of parameters are passed.

## Examples

The following example creates a deferred entry for program COT0, passing the string VPH as input data to the program.

```
#include <tpfapi.h>
void COT0();
char *parmstring = "VPH";
:
:
crexc(strlen(parmstring),parmstring,COT0);
```

## Related Information

- “credc, \_\_CREDC—Create a Deferred Entry” on page 57
- “creec, \_\_CREEC—Create a New ECB with an Attached Block” on page 59
- “cremc, \_\_CREMC—Create an Immediate Entry” on page 62
- “cretc, \_\_CRETCL—Create a Time-Initiated Entry” on page 64
- “cretc\_level, \_\_CRETCL—Create a Time-Initiated Entry” on page 66.

## croscc\_entrcc–Cross-Subsystem to Enter a Program and then Return

This function provides the ability to enter a program and then return to the calling program.

### Format

```
#include <sysapi.h>
long int croscc_entrcc(const char *segment, int idloc,
    struct TPF_regs *regs, ...);
```

#### segment

A pointer to the 4-character name of the segment to be entered by croscc\_entrcc. (It does not need to end with \0.)

#### idloc

Specifies the subsystem in which the segment is entered. Three values are defined for this parameter: CROSC\_BSS, CROSC\_DBI, and CROSC\_PBI. These values correspond to the assembler macro parameters BSS, DBI, and PBI. You can also code the 2-byte subsystem ID for this parameter; for example. X'FF00'.

#### regs

Points to a TPF\_regs structure or is NULL.

#### TARGET(TPF) only

If **regs** is not null, the enter linkage works the same as if the segment had been called with `#pragma linkage(name, TPF, N);`. If **regs** is null, `#pragma linkage(name, TPF, C)` enter linkage is run with the optional parameter list passed to the called segment.

#### ISO-C only

If **regs** is not NULL, the parameter list (**regs**) is passed along to the called segment. If **regs** is NULL, the optional parameter list is passed to the called segment. When **regs** is NULL and the called segment is a TPF BAL program, an additional NULL should be coded as the first and only optional parameter to indicate no TPF\_regs to enter/back.

... Is the parameter list that is passed to the cross-subsystem entered segment if **regs** is null. The called function (if prototyped) takes only integral (char, short, int, long, and enums), pointer, and double parameters. A function call cannot be one of the parameters. The optional parameters must match the types of the corresponding prototyped parameters.

### Normal Return

The croscc\_entrcc function returns the value that was returned by the cross-entered function, cast to a (long int). Only integral and pointer values can be returned.

### Error Return

Not applicable.

## Programming Considerations

- ENTRC to FACE or FACS is a special type of enter and cannot be used on the croscc\_entrc function. Attempting to croscc\_entrc to FACE or FACS results in a CTL-66 dump.
- If the specified **idloc** is not valid, not genmed, or not active, an OPR-000230 or 000231 dump with exit is taken. If the caller does not want a dump, the caller must validate the idloc value before calling croscc\_entrc.

## Examples

The following example executes the C language program CDEF in subsystem 0xF00F and then executes assembly language program WXYZ in the BSS.

```
#include <tpfeq.h>
#include <sysapi.h>

#pragma linkage(WXYZ,TPF,N)
#pragma linkage(CDEF,TPF,C)

extern void WXYZ(struct TPF_regs *regs);
extern char *CDEF(int, double, char *);

:
/*
/* call type C segment with 3 parameters: 1, 2.0 and "ABC" */
/*
char *c = (char *)croscc_entrc("CDEF", 0xF00F, NULL,1, 2.0, "ABC");

struct TPF_regs reg = { 0 };

reg.r1 = 4;
croscc_entrc("WXYZ", CROSC_BSS, &reg);
:
```

## Related Information

- “cebic\_goto\_bss—Change MDBF Subsystem to BSS” on page 25
- “cebic\_goto\_ssu—Change MDBF Subsystem” on page 27
- “cebic\_restore—Restore Previously Saved DBI and SSU IDs” on page 28
- “cebic\_save—Save the Current DBI and SSU IDs” on page 29.

## crusa—Free Core Storage Block If Held

This function tests the state of one or more entry control block (ECB) data levels or data event control blocks (DECBs) and releases the core block from any that are occupied.

### Format

```
#include <tpfapi.h>
void      crusa(int count, enum t_lvl level, ...);
```

or

```
#include <tpfapi.h>
void      crusa(int count, TPF_DECB *decb, ...);
```

#### count

An integer containing the number of ECB data level or DECB parameters included in the parameter list.

#### level

One of 16 possible values representing a valid ECB data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F).

#### decb

A pointer to a DECB.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- Core block reference words (CBRWs) are initialized for all ECB data levels or DECBs from which blocks were released.
- Specifying an incorrect ECB data level or DECB results in a system error with `exit`.
- Specifying both ECB data levels and DECBs in the same function call results in a system error with `exit`. You must specify either all DECBs or all ECB data levels in the same function call.
- Specifying an incorrect count of parameters results in a system error with `exit`.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example tests ECB data levels D0, D4, DA, and any specified DECBs for working storage blocks, and releases the blocks if present.

```
#include <tpfapi.h>
...
TPF_DECB *decb1, *decb2, *decb3;
```

## crusa

```
      :  
crusa(3,DA,D0,D4);    /* Clear levels D0, D4, DA */  
      :  
crusa(3, decb1, decb2, decb3);  
                        /* release storage block if held by DECB*/
```

## Related Information

- “levtest—Test Core Level for Occupied Condition” on page 300
- “relcc—Release Working Storage Block” on page 421.

See *TPF Application Programming* for more information about DECBs.



## csonc—Convert System Ordinal Number

This function converts a database ordinal number (a file address reference format (FARF) format address) to a 7-byte file address in the MMCCHHR format. The name MMCCHHR displays how the address is constructed:

**MM**    A halfword symbolic module number used to identify the physical device  
**CC**    A halfword cylinder address  
**HH**    A halfword head address  
**R**      A 1-byte record number.

The 5-byte CCHHR is the same as the record ID recorded in the count area on file.

Validity checking is done by this function on a FARF file address for the following:

- Mode compatibility
- The correct limits of the FARF address.

## Format

```
#include <tpfio.h>
unsigned int csonc(enum t_lvl level, MCHR_STRUCT_PTR file_addr);
```

or

```
#include <tpfio.h>
unsigned int csonc(TPF_FA8 *fa8, MCHR_STRUCT_PTR file_addr);
```

### level

One of 16 possible values representing a valid entry control block (ECB) data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This indicates which ECB file address reference word (FARW) should be initialized with the program's file address and record ID. The specified FARW must contain a FARF address.

### fa8

A pointer to an 8-byte file address.

### file\_addr

The 2-byte module (MM), cylinder (CC), and head (HH) numbers, and also the 1-byte record number (R) are returned to the file address.

## Normal Return

A nonzero value.

## Error Return

An integer value of zero.

## Programming Considerations

- Check the return condition for a zero result, which denotes that an incorrect FARF address was given as input to this function.
- The file address contains the 2-byte module (MM), cylinder (CC), and head (HH) numbers, as well as the 1-byte record (R) number. The rightmost (8th) byte of the file address contains one of the following hexadecimal device type codes.

### Code    Symbolic Device Type

**X'0C'**    DEVA

## csonc

**X'10'** DEVB

**X'14'** DEVC

**X'18'** DEVD.

- Applications that call this function using 8-byte file addresses instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

In the following example, FACS is called to calculate a FARF address and convert it to a MMCCHHR address.

```
#include <tpfeq.h>
#include <tpfio.h>

#pragma linkage(FACS,TPF,N)
extern void FACS(struct TPF_regs *);
:
:
MCHR_STRUCT mchr;
struct TPF_regs regs;

/* call FACS to calculate FARF address */
regs.r0 = 0;
regs.r6 = (long int)"#RECTYPE";
regs.r7 = (long int)&ecbptr()->celfa0;
FACS(&regs);

if (regs.r0 == 0)
{
    /* FACS error */
}
else
{
    /* call csonc to convert FARW to MMCCHHR */
    if (csonc(D0, &mchr))
    {
        /* success */
    }
    else
    {
        /* failure */
    }
}
```

In the following example, tpf\_fac8c is called to calculate a FARF address and convert it to a MMCCHHR address.

```
#include <tpfeq.h>
#include <tpfio.h>
:
:
MCHR_STRUCT mchr;
TPF_FA8      fileAddr;
TPF_FAC8     parms;

/* call tpf_fac8c to calculate FARF address */
memcpy(parms.ifacrec, "#RECTYPE", 8);
parms.ifacord=0;
parms.ifactyp=IFAC8FCS;
```

```
tpf_fac8c(&parms);  
  
if (parms.ifacret != TPF_FAC8_NRM)  
{  
    /* FACE/FACS error */  
}  
else  
{  
    fileAddr=parms.ifacadr;  
    /* call csonc to convert address to MMCCHHR */  
    if (csonc(&fileAddr, &mchr))  
    {  
        /* success */  
    }  
    else  
    {  
        /* failure */  
    }  
}
```

## Related Information

None.

## dbsac—Attach TPF Application Requester Database Support Structure

This function is used to attach the TPFAR Structured Query Language Transaction Profile (STP) and associated blocks to the existing ECB. This TPFAR database support structure must have been detached from this or another ECB using the `dbstdc.h` function.

### Format

```
#include <tpfarapi.h>
enum DBSAC_RETURN_CODE dbsac(char *id);
```

**id** A pointer to an 8-byte area that contains the TPF Application Requester (TPFAR) database support structure identifier. This 8-byte identifier must have been previously supplied by the `dbstdc.h` function.

### Normal Return

#### **DBSAC\_SUCCESSFUL**

Completed successfully.

### Error Return

#### **DBSAC\_INUSE**

A TPFAR database support structure is already attached to the current ECB.

#### **DBSAC\_DBSFINDERR**

Cannot find the TPFAR database support structure.

#### **DBSAC\_CCAFINDERR**

Cannot find a file record associated with a cursor control area (CCA) entry.

## Programming Considerations

- A system error with return results if an incorrect TPFAR database support structure identifier is specified.
- Once the TPFAR database support structure is reattached to the ECB, the TPFAR database support structure identifier is no longer valid and must not be reused.

## Examples

The following example reattaches a previously detached TPFAR database support structure.

```
#include <tpfarapi.h>
char id[8]
:
:
dbsac(id);
```

## Related Information

- “dbstdc—Detach TPF Application Requester Database Support Structure” on page 79
- For more information about TPFAR, see the *TPF Application Requester User's Guide*.

## dbstdc–Detach TPF Application Requester Database Support Structure

This function is used to detach the TPFAR Structured Query Language Transaction Profile (STP) and associated blocks from the ECB for later reattachment to this or another ECB.

### Format

```
#include <tpfarapi.h>
enum DBSDC_RETURN_CODE dbstdc(char *id);
```

**id** A pointer to an 8-byte area where the function places the TPF Application Requester (TPFAR) database support structure identifier.

### Normal Return

#### DBSDC\_SUCCESSFUL

Completed successfully.

### Error Return

#### DBSDC\_NONE

The TPFAR database support structure is not attached to the ECB.

## Programming Considerations

- Structured query language (SQL) requests must have been previously issued by this ECB.
- If the TPFAR database support structures are not reattached using the dbstdc function, conversations with the remote database are lost until the next IPL.
- The TPFAR database support structure identifier returned is needed to reattach the TPFAR database support structure later.
- The dbstdc.h function uses short-term pool files to save the database support structure. The dbstdc function must be issued before the short-term pool directories are recycled.

## Examples

The following example detaches the TPFAR database support structure from the ECB.

```
#include <tpfarapi.h>
char id[8]
:
dbstdc(id);
```

## Related Information

- “dbstdc–Attach TPF Application Requester Database Support Structure” on page 78
- For more information about TPFAR, see the *TPF Application Requester User’s Guide*.

---

## defrc—Defer Processing of Current Entry

This function is used to defer processing of an entry until the amount of activity in the system is low enough to allow for completion of the current low-priority task. The entry is placed on the deferred list, the lowest priority list in the CPU loop list schedule.

### Format

```
#include <tpfapi.h>
void defrc(void);
```

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- Following execution of this service, the ECB is added to the Defer List. Control is then transferred for processing another entry.
- The execution of this function resets the 500-millisecond program time-out.
- If a record is held by the ECB when this function is called, a system error (CTL-00D) with exit will result.

### Examples

The following example suspends processing of the currently executing program that is running now and places the associated ECB on the deferred list.

```
#include <tpfapi.h>
:
defrc();
```

### Related Information

- “credc, \_\_CREDC—Create a Deferred Entry” on page 57
- “dlayc—Delay Processing of Current Entry” on page 90
- “waitc—Wait For Outstanding I/O Completion” on page 686.

## deleteCache—Delete a Logical Record Cache

This function deletes a logical record cache from the processor.

### Format

```
#include <c$cach.h>
long      deleteCache( cacheToken *cache_to_release);
```

#### cache\_to\_release

The returned cache\_token from the newCache function that defined the logical record cache.

### Normal Return

#### CACHE\_SUCCESS

The function is completed successfully.

### Error Return

#### CACHE\_ERROR\_HANDLE

The cache\_token provided for the cache\_to\_release parameter is not valid.

## Programming Considerations

The delete occurs immediately and without regard for other applications that may also be using the cache.

## Examples

The following example deletes the file system INODE cache.

```
#include <c$cach.h>
#include <i$glue.h>

      struct icontrl * contrl_ptr;      /* pointer file system control area */

/* get pointer to file system control area */
      contrl_ptr = cinfc_fast_ss(CINFC_CMMZERO,
                                ecbptr()->ce1dbi );

/* if using INODE cache, call to delete it */

      if (contrl_ptr->icontrl_icacheToken.token1 != 0)
          deleteCache(&contrl_ptr->icontrl_icacheToken);
```

## Related Information

- “deleteCacheEntry—Delete a Cache Entry” on page 82
- “flushCache—Flush the Cache Contents” on page 198
- “newCache—Create a New Logical Record Cache” on page 376
- “readCacheEntry—Read a Cache Entry” on page 413
- “updateCacheEntry—Add a New or Update an Existing Cache Entry” on page 673.

---

# deleteCacheEntry–Delete a Cache Entry

This function deletes a cache entry.

## Format

```
#include <c$cach.h>
long      deleteCacheEntry(const cacheToken *cache_to_delete,
                           const void *primary_key,
                           const long *primary_key_length,
                           const void *secondary_key,
                           const long *secondary_key_length);
```

### cache\_to\_delete

The returned cache\_token from the newCache function that created the logical record cache.

### primary\_key

A pointer to a field that contains the value of the primary key.

### primary\_key\_length

The length of the primary key for a cache entry. The length specified must be equal to or less than the maximum value specified for the newCache function.

### secondary\_key

A pointer to a field that contains the value of the secondary key. If a secondary key was not specified for the newCache function, this field is ignored and will contain a NULL pointer.

### secondary\_key\_length

The length of the specified secondary key. The length specified must be equal to or less than the maximum value specified for the newCache function.

## Normal Return

### CACHE\_SUCCESS

The function is completed successfully.

## Error Return

One of the following:

### CACHE\_ERROR\_HANDLE

The cache\_token provided for the cache\_to\_delete parameter is not valid.

### CACHE\_ERROR\_PARAM

The value passed for one of the parameters is not valid.

## Programming Considerations

- To ensure data integrity, the caller must ensure that a locking mechanism is used when using the deleteCacheEntry function.
- Primary and secondary keys must exactly match the primary and secondary keys used to add the entry to the cache.
- The entry control block (ECB) must have the same database ID (DBI) as the ECB used to add the entry.
- If the target cache is a processor shared cache using CF, all other processors will have their copies of the deleted entry invalidated.



## Examples

The following example deletes a cache entry from the file system INODE cache if it exists.

```
#include <c$cach.h>
#include <i$glue.h>

struct icontrl * contrl_ptr;    /* pointer file system control area */
const long primaryKeyLgh = sizeof(ino_t); /* INODE number is key */
long nodeord;                  /* INODE number */

/* get pointer to file system control area */

contrl_ptr = cinfo_fast_ss(CINFO_CMMZERO,
                          ecptr()->ceidbi );

/* if using INODE cache, call to delete possible old entry for this INODE */

if (contrl_ptr->icontrl_icacheToken.token1 != 0)
    deleteCacheEntry(&contrl_ptr->icontrl_icacheToken,
                    &nodeord,
                    &primaryKeyLgh,
                    NULL,
                    NULL);
```

## Related Information

- “deleteCache—Delete a Logical Record Cache” on page 81
- “flushCache—Flush the Cache Contents” on page 198
- “newCache—Create a New Logical Record Cache” on page 376
- “readCacheEntry—Read a Cache Entry” on page 413
- “updateCacheEntry—Add a New or Update an Existing Cache Entry” on page 673.

## deqc–Dequeue from Resource

This function informs the control program that the entry control block (ECB) has ended with a shared resource. It is used with the enqc function.

### Format

```
include <tpfapi.h>
int deqc(char *deq_name,
          enum e_qual ssu_qual);
```

#### deq\_name

The address of an 8-byte area that contains the deqc resource name.

#### ssu\_qual

Subsystem user (SSU) qualification for the resource. The argument must belong to the enumerated type e\_qual, which is defined in the tpfapi.h header file. Use one of the following predefined terms:

##### QUAL\_U

Subsystem qualification applies. The resource name is subsystem unique and is qualified by the database index (DBI) value for the subsystem.

##### QUAL\_S

System wide qualification applies. The resource name is not subsystem unique. Any ECB issuing an enqc function with **QUAL\_S** will be enqueued on the same named resource. If two enqc functions are issued with the same resource name but different **ssu\_qual** values, two different resource names are assumed to exist. The deqc function must have the same **ssu\_qual** value as the enqc function.

### Normal Return

Integer value of zero.

### Error Return

A system error is issued if you are attempting to call deqc from a resource that will never be held by this ECB through the enqc function. The ECB will exit.

## Programming Considerations

- The resource must have been defined using the enqc function.

### Examples

The following example dequeues a shared resource with **ssu\_qual** defined as **QUAL\_S**.

```
#include <tpfapi.h>
char resource_name[8];
deqc(resource_name, QUAL_S);
```

### Related Information

“enqc–Define and Enqueue a Resource” on page 102.

## detac—Detach a Working Storage Block from the ECB

This function detaches a working storage block from an entry control block (ECB). The ECB must be holding a storage block on the specified level.

### Format

```
#include <tpfapi.h>
void      detac(enum t_lvl level);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this data level is the block to be detached.

### Normal Return

Void. The CBRW has been modified to indicate that no block is held.

### Error Return

Not applicable.

## Programming Considerations

- Specifying an invalid data level results in a system error with exit.
- Excessive use of this function can cause a depletion of working storage; therefore, its use should be carefully monitored.
- The detach functions (`detac`, `detac_ext`, and `detac_id`) and the attach functions (`attach`, `attach_ext`, and `attach_id`) push and pop working storage blocks to and from a list chained from the ECB. The `attach` function always attaches the block most recently detached from the specified data level. The `attach_ext` function attaches different blocks depending on how it is coded. The `attach_id` function attaches a working storage block with a matching FARW field to an ECB.

When coding attach and detach function calls, use the `attach` and `detac` function calls together, or the `attach_ext` and `detac_ext` function calls together, or the `attach_id` and `detac_id` function calls together.

## Examples

The following example detaches a working storage block from level D6.

```
#include <tpfapi.h>
:
:
detac(D6);
```

## Related Information

- “getcc—Obtain Working Storage Block” on page 248
- “relcc—Release Working Storage Block” on page 421
- “attach—Attach a Detached Working Storage Block” on page 19
- “attach\_ext—Attach a Detached Working Storage Block” on page 21
- “detac\_ext—Detach a Working Storage Block from the ECB” on page 86
- “attach\_id—Attach a Detached Working Storage Block” on page 23
- “detac\_id—Detach a Working Storage Block from the ECB” on page 88.

See *TPF Application Programming* for more information about DECBs.

## detac\_ext—Detach a Working Storage Block from the ECB

This function detaches a working storage block from an entry control block (ECB) data level or data event control block (DECB). The ECB must be holding a storage block on the specified ECB data level or DECB.

### Format

```
#include <tpfapi.h>
void detac_ext(enum t_lvl level, int ext);
```

or

```
#include <tpfapi.h>
void detac_ext(TPF_DECB *decb, int ext);
```

#### level

One of 16 possible values representing a valid ECB data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the ECB data level (0–F). The working storage block on this ECB data level is the block to be detached.

#### decb

A pointer to a DECB.

#### ext

Use one or more of the following choices to indicate which block is to be detached. If you code more than one term for **ext**, you must separate the terms with a plus sign (+).

##### DETAC\_USER\_DEFAULT

Indicates that the user is unspecified, which implies that the blocks detached from this ECB data level or DECB will be attached in a last-in-first-out (LIFO) order.

##### DETAC\_USER\_ACPDB

Indicates that the user is ACPDB, which implies that you need to save the FARW contents of the ECB data level or DECB being detached. The saved FARW contents will be used as a key to attach the block through the `attac_ext` function.

##### DETAC\_CHECK

Indicates that the ECB data level or DECB being detached should be checked to ensure there is a block to be detached. If there is no block, a CTL-0D2 system error will occur. If there is a block, it will be detached.

##### DETAC\_NOCHECK

Indicates that no check will be made to determine if there is a block to be detached. The block is detached with no check. If `DETAC_NOCHECK` is specified, the service routine will make the ECB data level or DECB reusable without validating that a block is held.

##### DETAC\_DEFAULT

Indicates that the defaults are `DETAC_USER_DEFAULT` and `DETAC_CHECK`.

### Normal Return

Void. The CBRW has been modified to indicate that no block is held.

### Error Return

Not applicable.

## Programming Considerations

- Specifying an incorrect ECB data level or DECB results in a system error with exit.
- Excessive use of this function can cause a depletion of working storage; therefore, its use should be carefully monitored.
- The detach functions (detac and detac\_ext) and the attach functions (attac and attac\_ext) push and pop working storage blocks to and from a list chained from the ECB. The attac function always attaches the block most recently detached from the specified data level. The attac\_ext function attaches different blocks depending on how it is coded. The attac\_id function attaches a working storage block with a matching FARW field to an ECB.

When you code attach and detach function calls, use the attac and detac function calls together or the attac\_ext and detac\_ext function calls together.

- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example makes ECB data level D6 reusable by detaching the storage block from level D6. There is no check made to see if a block is held.

```
#include <tpfapi.h>
:
:
detac_ext(D6,DETAC_NOCHECK);
```

The following example saves the working storage block on the DECB referenced by the *decb* variable.

```
#include <tpfapi.h>
TPF_DECB *decb;
:
:
detac_ext(decb,DETAC_NOCHECK);
```

## Related Information

- “attac—Attach a Detached Working Storage Block” on page 19
- “attac\_ext—Attach a Detached Working Storage Block” on page 21
- “attac\_id—Attach a Detached Working Storage Block” on page 23
- “detac—Detach a Working Storage Block from the ECB” on page 85
- “detac\_id—Detach a Working Storage Block from the ECB” on page 88
- “getcc—Obtain Working Storage Block” on page 248
- “relcc—Release Working Storage Block” on page 421.

See *TPF Application Programming* for more information about DECBs.

## detac\_id–Detach a Working Storage Block from the ECB

This function detaches a working storage block from an entry control block (ECB) data level or data event control block (DECB). The ECB must be holding a storage block on the specified ECB data level or DECB.

Save the file address reference word (FARW) contents of the ECB data level or DECB being detached. The saved FARW contents will be used as a key to attach the block through the `attac_id` function.

### Format

```
#include <tpfapi.h>
void detac_id(enum t_lvl level);
```

or

```
#include <tpfapi.h>
void detac_id(TPF_DECB *decb);
```

#### level

One of 16 possible values representing a valid ECB data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this data level is the block to be detached.

#### decb

A pointer to a DECB.

### Normal Return

Void. The CBRW has been modified to indicate that no block is held.

### Error Return

Not applicable.

## Programming Considerations

- Specifying an ECB data level or DECB that is not valid results in a system error with `exit`.
- Excessive use of this function can cause a depletion of working storage; therefore, its use should be carefully monitored.
- The detach functions (`detac`, `detac_ext`, and `detac_id`) and the attach functions (`attac`, `attac_ext`, and `attac_id`) push and pop working storage blocks to and from a list chained from the ECB. The `attac` function always attaches the block most recently detached from the specified ECB data level. The `attac_ext` function attaches different blocks depending on how it is coded. The `attac_id` function attaches a working storage block with a matching FARW field to an ECB.

When you code attach and detach function calls, use the `attac` and `detac` function calls together, or the `attac_ext` and `detac_ext` function calls together, or the `attac_id` and `detac_id` function calls together.

- `detac_id` is the functional equivalent of the ACPDB option of the `detac_ext` function.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.

## Examples

The following example makes data level D6 reusable by detaching the storage block from level D6. A block must be held and can only be retrieved by using the information in the FARW.

```
#include <tpfapi.h>
...
detac_id(D6);
```

The following example saves the working storage block on the DECB referenced by the *decb* variable.

```
#include <tpfapi.h>
TPF_DECB *decb;
...
detac_id(decb);
```

## Related Information

- “getcc—Obtain Working Storage Block” on page 248
- “relcc—Release Working Storage Block” on page 421
- “attac—Attach a Detached Working Storage Block” on page 19
- “detac—Detach a Working Storage Block from the ECB” on page 85
- “attac\_ext—Attach a Detached Working Storage Block” on page 21
- “detac\_ext—Detach a Working Storage Block from the ECB” on page 86
- “attac\_id—Attach a Detached Working Storage Block” on page 23.

See *TPF Application Programming* for more information about DECBs.

---

## dlayc—Delay Processing of Current Entry

This function is used to delay processing of an entry in order to allow processing of other entries. The current entry is placed on the input list.

### Format

```
#include <tpfapi.h>
void dlayc(void);
```

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- Following the execution of this service, the ECB is added to the input list. Control is then transferred for processing of another entry.
- The execution of this function resets the 500-millisecond program time-out.
- Records should not be held by the ECB when this function is called. Too many ECBs waiting for the record being held could result in the depletion of working storage blocks as ECBs are unable to complete waiting for the held record.

## Examples

The following example suspends processing of the program executing now and places the associated ECB on the input list.

```
#include <tpfapi.h>
:
dlayc();
```

## Related Information

- “waitc—Wait For Outstanding I/O Completion” on page 686
- “defrc—Defer Processing of Current Entry” on page 80.



## dllfree—Free the Supplied Dynamic Link Library (DLL)

This function frees the supplied DLL handle and logically deletes the DLL from memory if the handle was the last handle accessing the DLL.

### Format

```
#include <dll.h>
int dllfree(dllhandle* dllHandle);
```

#### **dllHandle**

A pointer to the **dllhandle** structure that is described in the `dll.h` header file.

### Normal Return

The `dllfree` function returns the following value when the requested service has been performed.

#### **Value    Meaning**

**0**        Successful

### Error Return

The `dllfree` function returns one of the following values and sets `errno` if the return code is not 0:

#### **Value    Meaning**

- 1**        The **dllHandle** supplied is NULL or **dllHandle** is inactive.
- 2**        There are no DLLs to be deleted.
- 3**        A logical delete was performed, but the DLL is not physically deleted because the DLCL use count is > 1 or there is an implicit reference to the DLL.

**Note:** DLCL is an IBM internal-use-only control block.

- 4**        No match is found for input **dllHandle**.
- 5**        C++ destructors are currently running for this DLL. A `dllfree` function is already in progress.

### Programming Considerations

- If a DLL is loaded implicitly, it cannot be deleted with the `dllfree` function. See the *C/C++ for MVS/ESA V3R2 Programming Guide* or *OS/390 C/C++ User's Guide* for more information about the implicit use of DLLs.
- DLLs that are loaded explicitly with the `dllload` function and are not freed with a corresponding call to the `dllfree` function are freed automatically when the entry control block (ECB) exits.

### Examples

The following example shows how to use the `dllfree` function to free the **dllHandle**, which is in DLL load module DLLB.

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

int DLLA (void);
int DLLA() {
    dllhandle *handle;
    char *name="DLLB";
```

## dllfree

```
int (*fptr1)(int);
int *ptr1_var1;
int rc=0;

handle = dllload(name);          /* call to stream DLL */
if (handle == NULL) {
    printf("failed on call to DLLB DLL\n");
    exit(-1);
}

fptr1 = (int (*)(int)) dllqueryfn(handle,"f1");
                                /* retrieving f1 function */
if (fptr1 == NULL) {
    printf("failed on retrieving f1 function\n");
    exit(-2);
}

ptr1_var1 = dllqueryvar(handle,"var1");
                                /* retrieving var1 variable */
if (ptr1_var1 == NULL) {
    printf("failed on retrieving var1 variable\n");
    exit(-3);
}

rc = fptr1(*ptr1_var1);          /* execute DLL function f1 */
(*ptr1_var1)++;                  /* increment value of var1 */

rc = dllfree(handle);            /* freeing handle to DLLB DLL */
if (rc != 0) {
    printf("failed on dllfree call\n");
}
return (0);
}
```

## Related Information

- “dllload—Load the DLL and Connect It to the Application” on page 93
- “dllqueryfn—Obtain a Pointer to a DLL Function” on page 94
- “dllqueryvar—Obtain a Pointer to a DLL Variable” on page 96.

## dllload—Load the DLL and Connect It to the Application

This function loads the dynamic link library (DLL) into memory (if it has not been loaded previously) and connects the DLL to the application program. The function that called the DLL receives a handle that uniquely identifies the requested DLL for subsequent explicit requests for that DLL.

A different handle is returned for each successful call to the `dllload`. A DLL is physically loaded only once even though there may be many calls to the `dllload` function.

### Format

```
#include <dll.h>
dllhandle* dllload(char * dllName);
```

#### **dllName**

A pointer to the character string ending with the NULL (\0) character identifying the 4-character name of the DLL load module to be loaded.

### Normal Return

The `dllload` function returns a unique handle that identifies the DLL when the call is successful.

### Error Return

When the call is not successful, the `dllload` function returns NULL and sets `errno`.

## Programming Considerations

None.

## Examples

The following example shows how to call C `dllload` functions from a simple C application.

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

int DLLA(void) {
    dllhandle *handle;
    char *name="DLLB";

    handle = dllload(name);
    if (handle == NULL) {
        printf("failed on dllload of DLLB DLL\n");
        exit (-1);
    }
}
```

## Related Information

- “`dllqueryfn`—Obtain a Pointer to a DLL Function” on page 94
- “`dllqueryvar`—Obtain a Pointer to a DLL Variable” on page 96
- “`dllfree`—Free the Supplied Dynamic Link Library (DLL)” on page 91.

## dllqueryfn—Obtain a Pointer to a DLL Function

This function obtains a pointer to a dynamic link library (DLL) function.

### Format

```
#include <dll.h>
void (* dllqueryfn(dllhandle *dllHandle, char *funcName)) ();
```

#### **dllHandle**

A pointer to the `dllhandle` structure that is described in the `dll.h` header file.

The **dllHandle** was obtained previously by a successful `dllload` function call.

#### **funcName**

A pointer to the character string ending with the NULL (\0) character string that represents the name of an exported function from the DLL.

### Normal Return

The `dllqueryfn` function returns a pointer to a function (**dllHandle**) that can be used to call the desired function in a DLL.

### Error Return

When the call is not successful, the `dllqueryfn` function returns NULL and sets `errno`.

## Programming Considerations

For C++ programs, use the external symbol name of the function instead of the function name itself for the **funcName** parameter. Because of C++ function overloading, a function name may not uniquely define a particular function. For example, in C++, `f1(arg1)` is a different function than `f1(arg1,arg2)`. To request the particular function in C++, use the external symbol name found in the external symbol cross-reference of a compiler listing.

## Examples

The following C language example shows how to use the `dllqueryfn` function to obtain a pointer to function `f1`, which is in DLL load module `DLLB`.

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

int main(void) {
    dllhandle *handle;
    char *name="DLLB";
    int (*fptr1)();

    handle = dllload(name);
    if (handle == NULL) {
        printf("failed on dllload of DLLB DLL\n");
        exit (-1);
    }

    fptr1 = (int (*)()) dllqueryfn(handle,"f1");
    if (fptr1 == NULL) {
        printf("failed on retrieving f1 function\n");
        exit (-2);
    }
}
```

## **Related Information**

- “**dllload**—Load the DLL and Connect It to the Application” on page 93
- “**dllqueryvar**—Obtain a Pointer to a DLL Variable” on page 96
- “**dllfree**—Free the Supplied Dynamic Link Library (DLL)” on page 91.

---

## dllqueryvar—Obtain a Pointer to a DLL Variable

This function obtains a pointer to a dynamic link library (DLL) variable.

### Format

```
#include <dll.h>
void* dllqueryvar(dllhandle *dllHandle, char *varName);
```

#### **dllHandle**

A pointer to the `dllhandle` structure that is described in the `dll.h` header file.

The **dllHandle** was obtained previously by a successful `dllload` function call.

#### **varName**

A pointer to the character string ending with the NULL ('\0') character string that represents the name of an exported variable from the DLL.

### Normal Return

The `dllqueryvar` function returns a pointer to a variable in the storage of the DLL if the call is successful.

### Error Return

When the call is not successful, the `dllqueryvar` function returns NULL and sets `errno`.

### Programming Considerations

None.

### Examples

The following example shows how to use the `dllqueryvar` function to obtain a pointer to variable `var1`, which is in DLL load module `DLLB`.

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

int main(void) {
    dllhandle *handle;
    char *name="DLLB";
    int *ptr1_var1;

    handle = dllload(name);
    if (handle == NULL) {
        printf("failed on dllload of DLLB DLL\n");
        exit (-1);
    }

    ptr_var1 = dllqueryvar(handle,"var1");
    if (ptr_var1 == NULL) {
        printf("failed on retrieving var1 variable\n");
        exit (-3);
    }
}
```

### Related Information

- “`dllload`—Load the DLL and Connect It to the Application” on page 93
- “`dllqueryfn`—Obtain a Pointer to a DLL Function” on page 94
- “`dllfree`—Free the Supplied Dynamic Link Library (DLL)” on page 91.

## dup—Duplicate an Open File Descriptor

This function duplicates an open file descriptor.

### Format

```
#include <unistd.h>
int dup(int fildes);
```

#### **fildes**

The open file descriptor to be duplicated.

This function returns a new file descriptor, which is the lowest-numbered descriptor that is available. The new file descriptor refers to the same open file as **fildes** and shares any locks that may be associated with **fildes**.

The following operations are equivalent:

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

For more information, see “fcntl—Control Open File Descriptors” on page 129.

### Normal Return

If successful, the dup function returns a new file descriptor.

### Error Return

If not successful, the dup function returns `-1` and sets `errno` to one of the following:

<b>EBADF</b>	<b>fildes</b> is not a valid open file descriptor.
<b>EMFILE</b>	The process has already reached its maximum number of open file descriptors.

## Programming Considerations

None.

## Examples

The following example duplicates an open file descriptor using the dup function.

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

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

main() {
    int fd;
    if ((fd = dup(0)) < 0)
        perror("&dupf error");
    else {
        print_inode(0);
        print_inode(fd);
        puts("The file descriptors are different but");
    }
}
```

## dup

```
        puts("they point to the same file.");  
        close(fd);  
    }  
}
```

### Output

The inode of fd 0 is 30  
The inode of fd 3 is 30  
The file descriptors are different but  
they point to the same file.

## Related Information

- “close—Close a File” on page 44
- “creat—Create a New File or Rewrite an Existing File” on page 54
- “dup2—Duplicate an Open File Descriptor to Another” on page 99
- “fcntl—Control Open File Descriptors” on page 129
- “open—Open a File” on page 380.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



## dup2–Duplicate an Open File Descriptor to Another

This function duplicates an open file descriptor to another.

### Format

```
#include <unistd.h>
int dup2(int fd1, int fd2);
```

#### **fd1**

The open file descriptor to be duplicated.

#### **fd2**

The duplicate file descriptor.

This function returns a file descriptor with a value that is not less than **fd2**. **fd2** now refers to the same file as **fd1** and the file that was previously referred to by **fd2** is closed. The following conditions apply:

- If **fd2** is less than 0 or greater than OPEN\_MAX, the dup2 function returns –1 and sets errno to EBADF.
- If **fd1** is a valid file descriptor and is equal to **fd2**, the dup2 function returns **fd2** without closing it.
- If **fd1** is not a valid file descriptor, the dup2 function fails and does not close **fd2**.
- If a file descriptor does not already exist, the dup2 function can be used to create one, a duplicate of **fd1**.

### Normal Return

If successful, the dup2 function returns **fd2**.

### Error Return

If not successful, the dup2 function returns –1 and sets errno to the following:

**EBADF**            **fd1** is not a valid file descriptor, or **fd2** is less than 0 or greater than OPEN\_MAX.

### Programming Considerations

None.

### Examples

The following example duplicates an open file descriptor using the dup2 function.

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

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

main() {
    int fd;
    char fn[]="DUP2.FILE";
    if ((fd = creat(fn, S_IWUSR)) < 0)
```

## dup2

```
        perror("creat() error");
    else {
        print_inode(fd);
        if ((fd = dup2(0, fd)) < 0)
            perror("dup2() error");
        else {
            puts("After dup2()...");
            print_inode(0);
            print_inode(fd);
            puts("The file descriptors are different but they");
            puts("point to the same file which is different than");
            puts("the file that the second fd originally pointed to.");
            close(fd);
        }
        unlink(fn);
    }
}
```

### Output

```
The inode of fd 3 is 3031
After dup2()...
The inode of fd 0 is 30
The inode of fd 3 is 30
The file descriptors are different but they
point to the same file which is different than
the file that the second fd originally pointed to.
```

## Related Information

- “close—Close a File” on page 44
- “creat—Create a New File or Rewrite an Existing File” on page 54
- “dup—Duplicate an Open File Descriptor” on page 97
- “fcntl—Control Open File Descriptors” on page 129
- “open—Open a File” on page 380.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## ecbptr–ECB Reference

This macro returns a pointer to the eb0eb structure (see c\$eb0eb.h). It provides access to data in the ECB.

### Format

```
#include <c$eb0eb.h>
struct eb0eb *ecbptr(void);
```

### Normal Return

Pointer of type struct eb0eb to the currently executing ECB.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example obtains a pointer to the record on level D2.

```
#include <c$am0sg.h>
#include <c$eb0eb.h>
struct am0sg *amsg;          /* Pointer to message block */
:
amsg = ecbptr()->ce1cr2;     /* Get base of record */
```

### Related Information

None.

## enqc—Define and Enqueue a Resource

This function defines a shared resource to the control program. It also controls access to the resource among entry control blocks (ECBs). This function is used with the deqc function.

### Format

```
include <tpfapi.h>
int      enqc(char          *enq_name,
                  enum e_wait enq_wait,
                  unsigned long timeout,
                  enum e_qual ssu_qual);
```

#### enq\_name

The address of an 8-byte area that contains the enqc resource name.

#### enq\_wait

Indicates whether the ECB is forced to wait. The argument must belong to the enumerated type `e_wait`, which is defined in the `tpfapi.h` header file. Use one of the predefined terms:

##### ENQ\_WAIT

If specified, the ECB is forced to wait before gaining control of the resource.

##### ENQ\_NOWAIT

If specified, the ECB will not give up control if the named resource is already in use by another ECB.

#### timeout

An integer specifying the number of seconds that the ECB expects to hold the resource. The value can be specified with a range of 0–65 535. The timeout does not start until there is another ECB waiting. If the specified time elapses, the ECB will exit with a system error. If 0 is specified, the timeout function is turned off.

#### ssu\_qual

Subsystem qualification for the resource. The argument must belong to the enumerated type `e_qual`, which is defined in the `tpfapi.h` header file. Use one of the following predefined terms:

##### QUAL\_U

Subsystem qualification applies. The resource name is subsystem user (SSU) unique and is qualified by the database index (DBI) value for the subsystem.

##### QUAL\_S

System wide qualification applies. The resource name is not subsystem unique. Any ECB that issues the `enqc` function with **QUAL\_S** will be enqueued on the same named resource. If two `enqc` functions are issued with the same resource name but different **ssu\_qual** values, two different resource names are assumed to exist. The `deqc` function must have the same **ssu\_qual** value as the `enqc` function.

### Normal Return

#### ENQ\_WAIT

Return code `ECB_WAITED` is returned if the resource was held by another ECB and the calling ECB has waited for the resource. Otherwise, return code 0 is returned.

**ENQ\_NOWAIT**

An integer value of zero is returned if the calling ECB gains control of the named resource. An integer value of 1 (RESOURCE\_IN\_USE) is returned if the resource is already in use. The calling ECB will not give up control, but the ECB will not own the resource.

**Error Return**

None.

**Programming Considerations**

- When finished with the resource, the ECB must issue a deqc function. If the ECB exits holding a resource, a system error is issued and the resource is freed.

**Examples**

The following example defines and tries to gain access to a shared resource with a timeout value of 250 seconds and an **ssu\_qual** of QUAL\_S. The ECB will not wait if the resource is in use by another ECB.

```
#include <tpfapi.h>
char    resource_name[8];
enqc(resource_name, ENQ_NOWAIT, 250, QUAL_S);
```

**Related Information**

“deqc–Dequeue from Resource” on page 84.

## entdc, \_\_ENTDC—Enter a Program and Drop Previous Programs

This function transfers control to a TPF segment without return, clearing all program nesting levels.

### Format

```
#include <tpfapi.h>
void entdc(void (*segname)(), struct TPF_regs *regs);
void __ENTDC(const char *segname, struct TPF_regs (regs);
```

#### segname

For entdc, a pointer to the external macro to be called. For \_\_ENTDC, a pointer to the name of the segment to be called. **segname** must map to a BAL segment name, a TARGET(TPF) segment name (or transfer vector), or a TPF ISO-C DLM name.

#### Migration to ISO-C consideration

The TARGET(TPF) entdc function can handle segment names that are #pragma mapped. The ISO-C entdc macro only handles #pragma mapped names if the first 4 characters are the same as the program name.

#### TARGET(TPF) restriction

Only calls to certain TPF API functions permit the use of function pointers. Pointers to functions are not supported in TARGET(TPF); their use will normally result in a compiler error.

#### regs

Treated as a pointer to struct TPF\_regs (defined in <tpfregs.h>) that contains 8 signed long integer members, r0–r7. These members are used to load general registers R0–R7 before passing control. If this argument is coded as **NULL**, no registers are loaded.

### Normal Return

Not applicable.

### Error Return

Not applicable.

### Programming Considerations

- The program called through **segname** must not issue a return. It must return control to the TPF system through `exit` or `abort`.
- The ISO-C stack is reinitialized. All other ISO-C environment states and data structures, including the heap, static storage, TCA and CTCA, are left in their current states by this function.

#### TARGET(TPF) restriction

All auto-storage blocks, stack blocks, and static blocks are released when this macro is called.

## Examples

The following example initializes the R1 slot in the TPF\_regs structure with the address of a message block and initializes the R2 slot with the address in the CBRW on level D8. The entdc macro then transfers control to program FMSG without return, passing the values in the indicated registers.

```
#include <tpfapi.h>
void FMSG();
struct TPF_regs *regs = (struct TPF_regs *)&(ecbptr()->ebx000);
char *msg_text;
:
regs->r1 = (long int)msg_text;
regs->r2 = (long int)ecbptr()->celcr8;
entdc(FMSG, regs);
```

## Related Information

- “entrc—Enter a Program with Expected Return” on page 106
- “TPF\_CALL\_BY\_NAME, TPF\_CALL\_BY\_NAME\_STUB—Call (Enter) a Program by Name” on page 562.

## entrc—Enter a Program with Expected Return

This macro transfers control to a TPF segment. The current active program remains held by the entry control block (ECB). The address of the next sequential instruction (NSI) in the current program is saved for an expected return.

### Format

```
#include <tpfapi.h>
void      entrc(const char *program, struct TPF_regs *regs);
```

#### program

Pointer to the name of the application program you want to enter. **program** must be the 4-character name of a BAL segment, a TARGET(TPF) segment, or a TPF ISO-C DLM.

#### regs

Treated as a pointer to struct TPF\_regs (defined in <tpfregs.h>) that contains 8 signed long integer members, r0–r7. When calling a BAL segment, the TPF control program enter routine uses these members to load general registers R0–R7 before passing control to the entered segment. If this argument is coded as **NULL**, no registers are loaded.

When calling a TARGET(TPF) segment or TPF ISO-C DLM, the value of the **regs** parameter is passed the same as any other C function parameter. Therefore, the called entry point address must have a type equivalent to (void (\*)(struct TPF\_regs \*)). The <tpfapi.h> header file provides 2 typedefs for declaring functions and pointers to functions of this type, TPF\_BAL\_FN and TPF\_BAL\_FN\_PTR, defined as:

```
typedef void TPF_BAL_FN(struct TPF_regs *);
typedef TPF_BAL_FN *TPF_BAL_FN_PTR;
```

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

This function cannot be called from either of the following:

- C++ language
- C language that is compiled using the dynamic link library (DLL) option.

## Examples

The following example looks up a table of program names and calls the appropriate program based on the index parameter.

```
#include <tpfapi.h>
#include <tpfregs.h>
const char (*getpgmtbl(void))[4];
void polymorph(int index, struct TPF_regs *regs)
{
    entrc(getpgmtbl()[index], regs);

    :
}
```



## Related Information

- “entdc, \_\_ENTDC—Enter a Program and Drop Previous Programs” on page 104
- “TPF\_CALL\_BY\_NAME, TPF\_CALL\_BY\_NAME\_STUB—Call (Enter) a Program by Name” on page 562.

---

## evinc-Increment Count for Event

This function increments the count for a count event after the event has been defined with the `evntc` function. An `evnwc` function cannot have been issued for the event before the `evinc` call.

### Format

```
#include <tpfapi.h>
int      evinc(struct ev0bk *evninf);
```

#### evninf

The `evinc` parameter block. For more information about the `evinc` parameter block, see `struct ev0bk`.

### Normal Return

Integer value of zero.

### Error Return

An integer value of 1 is returned if the event is not found.

## Programming Considerations

- The specified event must be a count event. An `evnwc` was not issued for the event. If either of these conditions is violated, a system error occurs and the ECB is exited.
- The maximum number of times the count for an event can be incremented is 32 767.
- Defined events in Multiple Database Function (MDBF) systems are subsystem unique. A return code of 1 occurs if the database ID (DBI) of the issuer is not the same as that of the subsystem that defined the event.

## Examples

The following example increments the count of the event pointed to by `event_ptr`.

```
#include <tpfeq.h>
struct ev0bk *event_ptr;
:
evntc(event_ptr, EVENT_CNT, 'N', 250, EVNTC_NORM);
evinc(event_ptr);
```

## Related Information

- “`evnqc`—Query Event Status” on page 109
- “`evntc`—Define an Internal Event” on page 111
- “`evnwc`—Wait for Event Completion” on page 113.

## evnqc—Query Event Status

This function queries the status of a named event. It is used with the `evntc` and `postc` functions.

### Format

```
#include <tpfapi.h>
int      evnqc (struct ev0bk *evninf, enum t_evn_typ type);
```

#### evninf

The `evnqc` parameter block. For more information about the `evnqc` parameter block, see `struct ev0bk` and `struct tpf_ev0bk_list_data`.

#### type

The type of event being completed. The argument must belong to the enumerated type `t_evn_typ` defined in `tpfapi.h`. Use one of the predefined terms:

##### EVENT\_MSK

for mask events.

##### EVENT\_CNT

for count events.

##### EVENT\_CB\_Dx

where *x* is a single hexadecimal digit (**0–F**) for core events.

##### EVENT\_LIST

for list events.

### Normal Return

One of the following:

- **EVNQC\_INC** is returned if the event has not completed.
  - If the event is count-type, `evninf->evnpstinf.evnbkcl` contains the remaining count value for the event.
  - If the event is list-type, `evninf->evnbklc` contains the count of data items in the list, `evninf->evnbkls` contains the size of each data item, and `evninf->evnbkli` contains the list of data items for the event. To determine if a specific data item has been posted, locate that data item in the list and check the value of the status flag field for that data item.
  - If the event is mask-type, `evninf->evnpstinf.evnbkml` contains the remaining mask value for the event.
  - `evninf->evnbkm2` contains the accumulated POST MASK 2 for the event.
  - `evninf->evnbke` contains the error indicator if an error has occurred, or zero if no error has occurred.
- **EVNQC\_COM** is returned if the event has completed.
- **EVNQC\_NFD** is returned if the event name is not found.

### Error Return

None.

### Programming Considerations

- The `evnqc` function returns any current information about an outstanding event but does not affect the state of the event. If the event has completed, an `evnwc` function is still required to cause the event to be deleted from the system.

## evnqc

- Any ECB is allowed to interrogate any existing event of its subsystem.
- For core block type events, only the evnbkm1, evnbkm2, and evnbke fields can be returned. The core block information is returned on the evnwc macro.
- For list-type events, the returned list of data items is the same as the created list. If any data items were posted with an error, the gross error indicator is set in evnbke. Check the status flag field for each data item in the list to determine the item or items that are in error.

## Examples

The following example queries the control program about a count type event that it has created and prints “event finished” to the terminal if the function does not return **EVNQC\_INC**.

```
#include <tpfeq.h>
struct ev0bk *event_ptr;
:
event_ptr->evnpstin.evnbc1=1;
evntc(event_ptr, EVENT_CNT, 'N', 250, EVNTC_NORM);
if (evnqc(event_ptr,EVENT_CNT) != EVNQC_INC)
    printf("event finished\n");
```

## Related Information

- “evinc—Increment Count for Event” on page 108
- “evntc—Define an Internal Event” on page 111
- “evnwc—Wait for Event Completion” on page 113
- “postc—Mark Event Element Completion” on page 395
- “tpf\_genlc—Generate a Data List” on page 599
- “tpf\_sawnc—Wait for Event Completion with Signal Awareness” on page 627.

## evntc—Define an Internal Event

This function defines to the control program a named event that can be waited on by this ECB and posted by other ECBs. This function is used with the `evnwc` and `postc` functions.

### Format

```
include <tpfapi.h>
int      evntc(struct ev0bk  *evninf,
               enum t_evn_typ evtyp,
               char          evn_name,
               int           timeout,
               enum t_state  evstat);
```

#### evninf

A pointer to the `evntc` parameter block. For more information about the `evntc` parameter block, see `struct ev0bk` and `struct tpf_ev0bk_list_data`.

#### evtype

The type of event being defined. The argument must belong to the enumerated type `t_evn_typ` defined in `tpfapi.h`. Use one of the predefined terms:

##### EVENT\_MSK

for mask events.

##### EVENT\_CNT

for count events.

##### EVENT\_CB\_Dx

where *x* is a single hexadecimal digit (**0–F**) for core events.

##### EVENT\_LIST

for list events.

A counter event is complete when the specified count becomes zero. The `postc` function decreases the event count by 1. A mask event is complete when the mask is completely reset. The `postc` function uses a 16-bit mask to reset the mask bits. Core block events are completed after the first `postc` function call. A list event is completed when all the data items have been posted.

#### evn\_name

Indicates if the caller has supplied the event name. If **Y** is specified, the event name is supplied by the caller in `evninf->evnbkn`. If **N** is specified, a unique event name is generated by the function processor and returned in `evninf->evnbkn`.

#### timeout

An integer specifying the number of seconds an ECB waits before the event is assumed to be in error. The time-out parameter is a value with a range of 0 to 65 535. If 0 is specified as the time-out, the event will not time out. Note that the C API for `EVNTC` differs from the assembler version because the time-out parameter needs to be coded.

#### evstate

Indicates whether the event is run in **NORM** state only or in all states. Code **EVNTC\_1052** if the event can run in all states or **EVNTC\_NORM** if the event can only run in **NORM** state.

### Normal Return

Integer value of zero.

**evntc**

## **Error Return**

An integer value of 1 is returned if the event name already exists.

## **Programming Considerations**

To ensure completion of the operation, call the evnwc function.

## **Examples**

The following example defines a count event to be named by the system with a time-out of 250 seconds, and capable of running in all system states.

```
#include <tpfeq.h>
struct ev0bk event_blk;
:
event_blk.evpstinf.evnbc1 = 1;
evntc(&event_blk, EVENT_CNT, 'N', 250, EVNTC_1052);
```

## **Related Information**

- “evinc—Increment Count for Event” on page 108
- “evnqc—Query Event Status” on page 109
- “evnwc—Wait for Event Completion” on page 113
- “postc—Mark Event Element Completion” on page 395
- “tpf\_genlc—Generate a Data List” on page 599
- “tpf\_sawnc—Wait for Event Completion with Signal Awareness” on page 627.

## evnwc—Wait for Event Completion

This function waits for the completion of a named event. It is used with the `evntc` and `postc` functions.

### Format

```
#include <tpfapi.h>
int      evnwc(struct ev0bk *evninf, enum t_evn_typ type);
```

#### evninf

A pointer to the `evnwc` parameter block. For more information about the `evnwc` parameter block, see `struct ev0bk` and `struct tpf_ev0bk_list_data`.

#### type

The type of event being completed. The argument must belong to the enumerated type `t_evn_typ`, defined in `tpfapi.h`. Use one of the predefined terms:

##### EVENT\_MSK

for mask events

##### EVENT\_CNT

for count events

##### EVENT\_CB\_Dx

where *x* is a single hexadecimal digit (**0-F**) for core events

##### EVENT\_LIST

for list events.

### Normal Return

An integer value of zero.

- If this is a core block event, the CBRW specified by the `type` parameter contains the core block passed by the `postc` macro.
- If the event is count-type, `evninf->evnpstinf.evnbkcl` contains the remaining count value for the event.
- If the event is list-type, `evninf->evnbkcl` contains the count of data items in the list, `evninf->evnbkls` contains the size of a data item, and `evninf->evnbkli` contains the list data items for the event.
- If the event is mask-type, `evninf->evnpstinf.evnbkml` contains the remaining mask value for the event.
- `evninf->evnbkm2` contains the accumulated POST MASK 2 for the event.
- `evninf->evnbke` contains the error indicator if an error has occurred, or zero if no error has occurred.

### Error Return

One of the following:

- An integer value of 1 is returned if the event name is not found.
- An integer value of 2 is returned if the event is an error post.

### Programming Considerations

- The `evnwc` function suspends the issuing ECB until the specified event completes, either by being posted or by timing out.
- Issuing the `evnwc` function causes an unconditional loss of control for the issuing ECB unless the named event is not found.

## evnwc

- More than 1 ECB is allowed to wait for the same named event except for core block events. When the event is completed, all waiting ECBs are posted. For a core block event, only the creating ECB can wait on the event. Any other ECB attempting to wait on the event is be returned with a not-found condition.

## Examples

The following example creates an event and waits for the event to complete.

```
#include <tpfeq.h>
struct ev0bk  event_blk;
enum t_evn_typ event_type;
char          caller_provided_name;
int           event_timeout;
enum t_state  event_state;
:
:
evntc(&event_blk, event_type, caller_provided_name, event_timeout,
      event_state);
evnwc(&event_blk, event_type);
```

## Related Information

- “evinc–Increment Count for Event” on page 108
- “evnqc–Query Event Status” on page 109
- “evntc–Define an Internal Event” on page 111
- “postc–Mark Event Element Completion” on page 395
- “tpf\_genlc–Generate a Data List” on page 599
- “tpf\_sawnc–Wait for Event Completion with Signal Awareness” on page 627.



## exit–Exit an ECB

This function ends a program.

### Format

```
#include <stdlib.h>
void exit(int return_code);
```

#### **return\_code**

Integer value indicating status.

#### **TARGET(TPF) restriction**

The `return_code` must be from 0 to 0x00FFFFFF. If nonzero, a system error bearing this identification number is issued before exiting.

Using the `exit` function is one way to cause *normal program termination*. (The other ways are calling the basic assembler language (BAL) `EXITC` macro and returning from the initial `main` function.) During normal program termination:

1. All functions registered by the `atexit` function are called in last-in-first-out (LIFO) order.
2. All open file streams and all open file descriptors are closed.

**Note:** Sockets are not closed and must be closed explicitly by the application.

3. All files created by the `tmpfile` function are deleted.
4. Control returns to the host environment.
5. If the entry control block (ECB) was created by a call to the `system` function, the `tpf_cresc()` function, or the BAL `CRESC` macro, the **return\_code** value returns to the parent program, which resumes running after the child ECB has exited. Otherwise, the TPF system ignores the **return\_code** value.

#### **TARGET(TPF) restriction**

Coding a nonzero return code as the argument will result in a system error dump.

### Normal Return

The `exit` function does not return to its caller. This function gives control to the TPF system, which exits the ECB. If the ECB was created by a parent ECB through the `system` or `tpf_cresc` functions, or the BAL `CRESC` macro, the TPF system reactivates the parent ECB. For example, if program A calls program B through a call to the `system` function, and program B calls the `exit` function, program B exits and program A resumes running at the next sequential instruction (NSI).

### Error Return

Not applicable.

### Programming Considerations

- Following the call to the `exit` function, no return is made to the operational program.

## exit

- If the `exit` function is called in a commit scope, control is transferred to the system error routine and processing ends as if a rollback was issued. The DASD surface remains unchanged.
- The return code parameter is used only if the exiting program was created using an ECB create macro such as `tpf_cresc`. In this case, the parent ECB is currently waiting for an event to be completed. When all child ECBs that were created by this parent ECB have exited, the event has completed successfully and control returns to the parent ECB.

### **TARGET(TPF) only consideration**

Coding a nonzero argument results in a system error with `exit`.

### **TARGET(TPF) only consideration**

The macros `EXIT_SUCCESS` and `EXIT_FAILURE` are provided for compatibility with other implementations, but their use is not recommended for TPF systems.

### **TARGET(TPF) only consideration**

If the **return\_code** parameter is less than zero or greater than `X'00FFFFFF'`, a system error with `exit` is generated (system error number that is not valid passed to `exit` function).

### **TARGET(TPF) only consideration**

No prefix character is concatenated with the system error number. Therefore, there is no way to distinguish between IBM and user `exit` calls that code the same system error number.

## Examples

The following example exits the ECB when processing is completed. No system error is issued.

```
#include <stdlib.h>
int main(void)
{
    ...
    exit(0);
}
```

## Related Information

- “abort—Terminate Program Abnormally” on page 7
- “system—Execute a Command” on page 527.

## FACE, FACS–Low-Level File Address Compute Functions

This function initializes a FARW with the data necessary to access a fixed file record.

### Format

```
#include <tpfio.h>
void FACE(struct TPF_regs *regs);
void FACS(struct TPF_regs *regs);
```

#### regs

A pointer to a TPF\_regs structure (as defined in <tpfregs.h>). Members r0, r6 and r7 of the regs structure must be set up as follows:

**r0** Ordinal number of record.

**r6** Record type value for FACE() function; address of 8-character FACE ID for FACS() function.

**r7** Address of FARW for file address on output.

### Normal Return

regs.r0 contains the maximum ordinal number for the requested record type. regs.r6 and regs.r7 are destroyed. The file address has been placed in the FARW specified on input by regs.r7.

### Error Return

regs.r0 contains zero. regs.r7 contains a general error type indicator, and regs.r6 contains a specific error type indicator within the general type indicated by regs.r7:

#### regs.r7 == 1

The record type indicator is invalid.

#### regs.r6 == 1

The requested record type is not in use.

#### regs.r6 == 2

The record type is not defined or exceeds the limit.

#### regs.r6 == 3

There are no records defined for the record type for the associated SSU, processor and I-stream.

#### regs.r7 == 2

The input ordinal number is outside allowable range.

#### regs.r6 == 0

The ordinal number exceeds the record type limit.

#### regs.r6 != 0

The requested ordinal does not exist but does not exceed the maximum. regs.r6 contains the next valid ordinal following the requested one. (This return is valid for pools with PSON gaps).

### Programming Considerations

- User macros that define equates for record types can be converted to C header files by the C/370 DSECT Conversion Utility, described in *SAA AD/Cycle C/370 User's Guide*. For example, if user record type equates are defined by macro MYRECIDS, creating the following assembler file:

## FACE, FACS

```
MYRECIDS CSECT ,
NOLABEL DS F           The utility requires a field to in order \
                        to create a structure.
MYRECIDS ,           The EQUs will be converted to #defines.
END ,
```

and processing it using the DSECT Conversion Utility with parameters: EQU(DEF) NOLC SEQ UNN will produce a C source file containing a structure (for NOLABEL) and #defines for all of the equates in MYRECIDS, consistent with the record type #defines in <c\$ysequ.h>.

## Examples

The following example generates a SON address for '#PROG1' record number 235 and stores it in the FARW for data level D6.

```
#include <tpfio.h>
:
:
struct TPF_regs regs;
regs.r0 = 235;
regs.r6 = (long int)"#PROG1 ";
regs.r7 = (long int)&ecbptr()->celfa6;
FACS(&regs);
if (regs.r0 != 0)
{
    /* Success, regs.r0 contains the maximum record ordinal for */
    /* for #PROG1 fixed file records, celfa6 contains the file */
    /* address. */
    :
}
else /* Determine type of error */
{
    switch (regs.r7)
    {
    case 1: /* The record type indicator is invalid. */
        switch (regs.r6)
        {
        case 1:
            /* The requested record type is not in use. */
            break;
        case 2:
            /* The record type is not defined or exceeds the limit. */
            break;
        case 3:
            /* There are no records defined for the record type for */
            /* the associated SSU. */
            break;
        }
        break;
    case 2: /* The input ordinal number is outside allowable range. */
        if (regs.r6 == 0)
        {
            /* The ordinal number exceeds the record type limit. */
        }
        else
        {
            /* The requested ordinal does not exist but does not */
            /* exceed the maximum. regs.r6 contains the next valid */
            /* ordinal following the requested one. (This return is */
            /* valid for pools with PSON gaps). */
        }
    }
}
```

## **Related Information**

"face\_facs—File Address Generation" on page 120.

## face\_facs—File Address Generation

This function initializes a FARW with the data necessary to access a fixed file record.

### Format

```
#include <tpfio.h>
int face_facs(unsigned long record_ordinal, const char *record_type_name,
              unsigned long record_type_id, enum t_lvl level,
              unsigned long *rtnord);
```

#### record\_ordinal

The ordinal number of the fixed file record.

#### record\_type\_name

A string containing the name of the fixed file record type. It may optionally be padded to the right with blanks. If this parameter is coded as NULL, the record\_type\_id parameter is used to determine the fixed file record type.

#### record\_type\_id

The record type ID number for the fixed file record. This parameter is only used to determine the fixed file record type when record\_type\_name is coded as NULL; if record\_type\_name is not NULL, record\_type\_id should be coded as zero.

#### level

One of 16 possible values representing a valid data level from enumeration type t\_lvl, expressed as **Dx**, where x represents the hexadecimal number of the level (0–F). This parameter specifies the file address reference word (FARW) into which the SON address corresponding to the specified record ordinal and record type will be placed. Once modified by this function, the contents of this data level should not be altered by the application.

#### rtnord

The address of an unsigned long into which, on successful completion, the face\_facs() function stores the maximum ordinal number for the specified record type. If face\_facs() returns with the error condition

**FACE\_FACS\_RECORD\_ORDINAL\_GAP**, the next valid ordinal after record\_ordinal is returned in rtnord.

### Normal Return

Zero.

### Error Return

#### FACE\_FACS\_RECORD\_TYPE\_NOT\_IN\_USE

The record\_type\_name or record\_type\_id value passed to the face\_facs() function is not valid because the requested record type is not in use.

#### FACE\_FACS\_INVALID\_RECORD\_TYPE

The record\_type\_name or record\_type\_id value passed to the face\_facs() function is not valid because the requested record type is not defined or it exceeds the limit.

#### FACE\_FACS\_RECORD\_TYPE\_EMPTY

The record\_type\_name or record\_type\_id value passed to the face\_facs() function is not valid because there are no records defined for the requested record type in the associated SSU.

**FACE\_FACS\_INVALID\_RECORD\_ORDINAL**

The record\_ordinal value passed to the face\_facs() function is not valid for the specified record type because it exceeds the record type limit.

**FACE\_FACS\_RECORD\_ORDINAL\_GAP**

The record\_ordinal value passed to the face\_facs() function is not valid for the specified record type, but it does not exceed the record type limit. rtnord contains the next valid ordinal after record\_ordinal. (This return can occur for pools with PSON gaps.)

**FACE\_FACS\_INVALID\_DATA\_LEVEL**

The level value passed to the face\_facs function is not a valid data level (D0-DF).

## Programming Considerations

For information about converting record type equates in assembler to C #defines, see "Programming Considerations" on page 117.

## Examples

The following example generates a SON address for "#PROG1" record number 235 and stores it in the FARW for data level D6.

```
#include <tpfio.h>
:
unsigned long prog1_ordinal;
int rc = face_facs(235, "#PROG1", 0, D6, &prog1_ordinal);
switch (rc)
{
    case 0: /* Success: the FARW at data level D6 contains the */
            /* file address for "#PROG1" record ordinal number */
            /* 235, and prog1_ordinal contains the maximum record */
            /* ordinal for #PROG1 fixed file records. */
            :
            break;
    case FACE_FACS_RECORD_TYPE_NOT_IN_USE:
    :
    break;
    case FACE_FACS_INVALID_RECORD_TYPE:
    :
    break;
    case FACE_FACS_RECORD_TYPE_EMPTY:
    :
    break;
    case FACE_FACS_INVALID_RECORD_ORDINAL:
    :
    break;
    case FACE_FACS_RECORD_ORDINAL_GAP:
        /* prog1_ordinal contains the next valid ordinal after */
        /* 235. */
        :
    break;
    case FACE_FACS_INVALID_DATA_LEVEL:
    :
    break;
}
```

## Related Information

- “FACE, FACS—Low-Level File Address Compute Functions” on page 117
- “filec—File a Record: Basic” on page 152
- “filec\_ext—File a Record with Extended Options: Basic” on page 154
- “file\_record—File a Record: Higher Level” on page 158
- “file\_record\_ext—File a Record with Extended Options: Higher Level” on page 161
- “filnc—File a Record with No Release” on page 165
- “filnc\_ext—File a Record with No Release and Extended Options” on page 167
- “filuc—File and Unhold a Record” on page 170
- “filuc\_ext—File and Unhold a Record with Extended Options” on page 172
- “findc—Find a Record” on page 174
- “findc\_ext—Find a Record with Extended Options” on page 176
- “find\_record—Find a Record” on page 178
- “find\_record\_ext—Find a Record with Extended Options” on page 181
- “finhc—Find and Hold a File Record” on page 185
- “finhc\_ext—Find and Hold a File Record with Extended Options” on page 187
- “finwc—Find a File Record and Wait” on page 189
- “finwc\_ext—Find a File Record and Wait with Extended Options” on page 191
- “fiwhc—Find and Hold a File Record and Wait” on page 193
- “fiwhc\_ext—Find and Hold a File Record and Wait with Extended Options” on page 195.



## fchmod—Change the Mode of a File or Directory by Descriptor

This function changes the mode of a file or directory by a file descriptor.

### Format

```
#include <sys/stat.h>
int fchmod(int fildes, mode_t);
```

#### fildes

An open file descriptor for the file or directory whose permissions are to be changed.

#### mode

The new permission. The **mode** argument is created with one of the symbols defined in the `sys/stat.h` header file. See “chmod—Change the Mode of a File or Directory” on page 32 for information about these symbols and the **mode** parameter.

This function sets the file permission bits of the open file identified by **fildes**, its file descriptor.

A process can set mode bits only if the effective user ID of the process is the same as the file’s owner or if the process has appropriate privileges (superuser authority). The `chmod` function automatically clears the `S_ISGID` bit in the file’s mode bits if all the following conditions are true:

- The calling process does not have appropriate privileges, that is, superuser authority (UID=0).
- The group ID of the file does not match the effective group ID of the calling process.
- One or more of the `S_IXUSR`, `S_IXGRP`, or `S_IXOTH` bits of the file mode are set to 1.
- The file is a regular file.

#### TPF deviation from POSIX

The TPF system does not support the `ctime` time stamp.

### Normal Return

If successful, the `fchmod` function returns 0.

### Error Return

If unsuccessful, the `fchmod` function returns `-1` and sets `errno` to one of the following:

<b>EBADF</b>	<b>fildes</b> is not a valid open file descriptor.
<b>EPERM</b>	The user ID (UID) does not match the owner of the file.

### Programming Considerations

None.

### Examples

The following example changes a file permission.

## fchmod

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

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

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

### Output

```
original permissions were: 03000000
after fchmod(), permissions are: 030001f8
```

## Related Information

- “chmod—Change the Mode of a File or Directory” on page 32
- “chown—Change the Owner or Group of a File or Directory” on page 35
- “fchown—Change the Owner and Group by File Descriptor” on page 125
- “mkdir—Make a Directory” on page 326
- “open—Open a File” on page 380.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## fchown—Change the Owner and Group by File Descriptor

This function changes the owner and group by a file descriptor.

### Format

```
#include <unistd.h>
int fchown(int fildes, uid_t owner, gid_t group);
```

#### fildes

An open file descriptor for the file or directory whose owner is to be changed.

#### owner

The new user ID (UID) for the file or directory.

#### group

The new group ID (GID) for the file or directory.

A process can change the group of a file only if the effective user ID of the process is equal to the user ID of the file owner or if the process has appropriate privileges (superuser authority). If the file is a regular file and the process does not have superuser authority, the chown function clears the **S\_ISUID** and **S\_ISGID** bits in the mode of the file.

#### TPF deviation from POSIX

The TPF system does not support the ctime time stamp.

### Normal Return

If successful, the fchown function updates the owner and group for the file and returns 0.

### Error Return

If unsuccessful, the fchown function returns -1 and sets errno to one of the following:

**EBADF**            **fildes** is not a valid open file descriptor.

**EPERM**            The user ID does not match the owner of the file.

### Programming Considerations

None.

### Examples

The following example changes the owner ID and group ID.

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

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

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
```

## fchown

```
else {
    stat(fn, &info);
    printf("original owner was %d and group was %d\n", info.st_uid,
           info.st_gid);
    if (fchown(fd, 25, 0) != 0)
        perror("fchown() error");
    else {
        stat(fn, &info);
        printf("after fchown(), owner is %d and group is %d\n",
               info.st_uid, info.st_gid);
    }
    close(fd);
    unlink(fn);
}
```

### Output

```
original owner was 256 and group was 257
after fchown(), owner is 25 and group is 0
```

## Related Information

- “chown—Change the Owner or Group of a File or Directory” on page 35
- “chmod—Change the Mode of a File or Directory” on page 32
- “fchmod—Change the Mode of a File or Directory by Descriptor” on page 123
- “mkdir—Make a Directory” on page 326
- “open—Open a File” on page 380.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## fclose—Close File Stream

This function closes a file stream.

### Format

```
#include <stdio.h>
int fclose(FILE *stream);
```

#### stream

A pointer to the open file stream to be closed.

This function flushes a stream and then closes the file associated with that stream. The function then releases any buffers associated with the stream. To flush means that unwritten buffered data is written to the file and buffered data that is not read is discarded.

**Note:** The storage pointed to by the FILE pointer is freed by the `fclose` function. An attempt to use the FILE pointer to a closed file is incorrect. This restriction is true even when the `fclose` function fails.

A pointer to a closed file **cannot** be used as an input value to the `freopen` function.

### Normal Return

If successful, the `fclose` function returns 0.

### Error Return

If a failure occurs when flushing buffers or in outputting data, EOF is returned. An attempt is still made to close the file.

The `fclose` function sets `errno` to one of the following:

<b>EAGAIN</b>	The <code>O_NONBLOCK</code> flag is set and output cannot be written immediately.
<b>EBADF</b>	The underlying file descriptor is not valid.
<b>EFBIG</b>	Writing to the output file would exceed the maximum file size or the file size of the process supported by the implementation.
<b>ENOSPC</b>	There is no free space left on the output device

## Programming Considerations

None.

## Examples

The following example opens the file `myfile.dat` for reading as a stream and then closes the file.

```
#include <stdio.h>

int main(void)
{
    FILE *stream;

    stream = fopen("myfile.dat", "r");
    :
    if (fclose(stream)) /* Close the stream. */
        printf("fclose error\n");
}
```

**fclose**

## **Related Information**

- “fopen—Open a File” on page 199
- “freopen—Redirect an Open File” on page 215.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## fcntl—Control Open File Descriptors

This function controls the open file descriptors.

### Format

```
#include <fcntl.h>
int fcntl(int fildes, int action, ...);
```

#### fildes

A file descriptor for the file you want to change.

#### action

A symbol indicating the action you want to perform. These symbols are defined in the `fcntl.h` header file.

The **action** argument can be one of the following symbols:

#### F\_DUPFD

Duplicates the file descriptor. A third `int` argument must be specified. The `fcntl` function returns the lowest file descriptor greater than or equal to this third argument that is not already associated with an open file. This file descriptor refers to the same file as **fildes** and shares any locks. The `FD_CLOEXEC` flag is turned off in the new file descriptor so that the file descriptor is inherited by a process activated by the system function.

#### F\_GETFD

Obtains the file descriptor flags for **fildes**. The `fcntl` function returns these flags as its result.

#### F\_SETFD

Sets the file descriptor flags for **fildes**. You must specify a third `int` argument, giving the new file descriptor flag settings. The `fcntl` function returns 0 if it successfully sets the flags.

#### F\_GETFL

Obtains the file status flags, file access mode flags, and the `O_TPF_NODDCLOSE` for **fildes**. `fcntl` returns these flags as its result.

#### F\_SETFL

Sets the file status flags for **fildes**. You must specify a third `int` argument, giving the new file descriptor flag settings. `fcntl` does not change the file access mode, and file access bits in the third argument are ignored. `fcntl` returns 0 if it successfully sets the flags.

#### F\_SETLKW

Sets or clears a file segment lock, but if a lock is blocked by other locks, `fcntl` waits until the request can be satisfied.

#### TPF deviation from POSIX

The TPF system does not support the `F_GETLK` and `F_SETLK` **action** parameter values.

... Indicates a third argument. The type of the third argument depends on **action**, and some actions do not need an additional argument.

Performs various actions on open file descriptors.

### File Flags

There are several types of flags associated with each open file. Flags for a file are represented by symbols defined in the `fcntl.h` header file.

The following *file descriptor* flag can be associated with a file:

**FD\_CLOEXEC** If this flag is 1, the file descriptor is closed in any child processes created by system function calls. If the flag is 0, a child process created by a system function call inherits a copy of the open file descriptor.

The following *file status* flags can be associated with a file:

**O\_APPEND** Append mode. If this flag is 1, every write operation on the file begins at the end of the file.

**O\_NONBLOCK** No blocking. If this flag is 1, read and write operations on the file return with an error status if they cannot perform their I/O immediately. If this flag is 0, read and write operations on the file wait (or **block**) until the file is ready for I/O. For more information, see “read—Read from a File” on page 410 and “write—Write Data to a File Descriptor” on page 696.

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

**O\_RDONLY** The file is opened for reading only.

**O\_RDWR** The file is opened for reading and writing.

**O\_WRONLY** The file is opened for writing only.

Two masks can be used to extract flags:

**O\_ACCMODE** Extracts file access mode flags.

**O\_GETFL** Extracts file status flags and file access mode flags.

The following *TPF-specific control* flag can be associated with a file:

**O\_TPF\_NODDCLOSE** This flag is set to on when the application needs to take control of the underlying system resources that are referenced by an open file description. It prevents the device driver `TPF_FSDD_CLOSE`-type function from being called when the last file descriptor referencing the open file description is closed.

## File Locking

### TPF deviation from POSIX

The only file locking supported by the TPF system is a blocking exclusive lock over an entire file. The `fcntl` function options to lock a range of bytes in a file (other than the entire file), to attempt to lock without blocking, to get a shared lock, and to query locks are not supported.

A process can use the `fcntl` function to lock out other processes from a file so that the process can read or write to that part of the file without interference from other



files. File locking can ensure data integrity when several processes have a file accessed concurrently. File locking can be performed only on file descriptors that refer to regular files. Locking is not permitted on file descriptors that refer to directories, character special files, or any other type of files.

A structure that has the type `struct flock` (defined in the `fcntl.h` header file) controls locking operations. This structure has the following members:

**short l\_type** Indicates the type of lock by using one of the following symbols (defined in the `fcntl.h` header file):

**F\_WRLCK** Indicates a *write lock* (also called an *exclusive lock*). The process has exclusive access to the locked file and no other process can establish a write lock on that same file. If another process already has a write lock on the file, this process waits until the file is free.

**TPF deviation from POSIX**

To allow a reader to get exclusive access to a file, the requirement to have write access to the file when requesting a write lock is not enforced.

**F\_UNLCK** Unlocks a lock that was set previously.

**TPF deviation from POSIX**

The TPF system does not support read or shared locks. The `fcntl` function does not accept the `F_RDLCK` lock type.

**short l\_whence**

The only accepted value is:

**SEEK\_SET** The start of the file.

This symbol is defined in the `unistd.h` header file and is the same as used by the `lseek` function.

**TPF deviation from POSIX**

The TPF system does not accept the `SEEK_CUR` and `SEEK_END` values for this field.

**off\_t l\_start** The only value accepted by the TPF system is 0, indicating that the lock begins at the beginning of the file.

**off\_t l\_len** The only value accepted by the TPF system is 0, indicating that the lock extends to the end of the file.

**TPF deviation from POSIX**

The TPF system does not support the `pid_t l_pid` field of the `flock` structure.

## fcntl

You can set a lock by specifying `F_SETLKW` as the **action** argument for the `fcntl` function. Such a function call requires a third argument pointing to a `struct flock` structure as in the following example:

```
struct flock lock_it;
lock_it.l_type = F_WRLCK;
lock_it.l_whence = SEEK_SET;
lock_it.l_start = 0;
lock_it.l_len = 0;
fcntl(filides, F_SETLKW, &lock_it);
```

The previous example requests an exclusive lock for the entire file, which is the only type of lock that the TPF system supports. You can unlock this lock by setting `l_type` to `F_UNLCK` and making the same call. If an `F_SETLKW` operation cannot set a lock, it waits until the lock can be set. For example, if you want to establish a lock and some other process already has a lock established on the same file, `fcntl` waits until the other process has removed its lock.

Deadlocks can occur with `F_SETLKW` operations when process A is waiting for process B to unlock a file, and process B is waiting for process A to unlock a different file. Applications that lock more than one file at a time need to establish locking conventions that prevent more than one process from deadlocking.

A process lock on a file is removed when the process closes any file descriptor that refers to the locked file. Locks are not inherited by child processes created with `system`.

All locks are advisory only. Processes can use locks to inform each other that they want to protect a file, but locks do not prevent I/O on the locked files. If a process has appropriate permissions on a file, it can perform whatever I/O it chooses regardless of what locks are set. Therefore, file locking is only a convention, and it works only when all processes respect the convention.

## Normal Return

If successful, the value returned will depend on the **action** that was specified.

## Error Return

If unsuccessful, `fcntl` returns `-1` and sets `errno` to one of the following:

<b>EBADF</b>	<b>filides</b> is not a valid open file descriptor.
<b>EINVAL</b>	In an <code>F_DUPFD</code> operation, the third argument is negative or greater than or equal to <code>OPEN_MAX</code> , which is the highest file descriptor value allowed for the process.  In a locking operation, <b>filides</b> refers to a file with a type that does not support locking, or the <code>struct flock</code> pointed to by the third argument has an incorrect form.
<b>EMFILE</b>	In an <code>F_DUPFD</code> operation, the process has already reached its maximum number of file descriptors or there are no available file descriptors greater than the specified third argument.
<b>ENOLCK</b>	In a <code>F_SETLKW</code> operation, the specified file is already locked by the requesting process.

## Programming Considerations

None.

## Examples

The following example opens a file and requests an exclusive lock for the entire file.

**Note:** This is the only type of locking supported by the TPF system.

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

int main(void)
{
    int fd = open("my.file", O_RDWR);

    if (fd < 0)
    {
        perror("open failure");
    }
    else
    {
        struct flock lock_it;

        lock_it.l_type = F_WRLCK;
        lock_it.l_whence = SEEK_SET;
        lock_it.l_start = 0;
        lock_it.l_len = 0;
        fcntl(fd, F_SETLKW, &lock_it);

        :
        close(fd)    /* close releases the lock */
    }
}
```

## Related Information

- “dup—Duplicate an Open File Descriptor” on page 97
- “dup2—Duplicate an Open File Descriptor to Another” on page 99
- “fsync—Write Changes to Direct Access Storage” on page 232.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## FD\_CLR–Remove File Descriptor from the File Descriptor Set

This function removes a file descriptor from the file descriptor set.

### Format

```
#include <sys/time.h>
void FD_CLR(int fd, fd_set* fdset)
```

**fd** The file descriptor.

**fdset**  
The file descriptor set.

### Normal Return

Void.

### Error Return

Void.

### Programming Considerations

- If you want to monitor more than 256 file descriptors on a single `tpf_select_bsd` call, the application must define its own `FD_SETSIZE` before the include of header `sys/time.h`.
- The purpose of the FD functions is to set up the input for and check the output from a `tpf_select_bsd` call. The `FD_CLR`, `FD_COPY`, `FD_SET`, and `FD_ZERO` functions set up the input for the `tpf_select_bsd` call. The `FD_ISSET` function checks the output from the `tpf_select_bsd` call.

### Examples

See the example provided for the `tpf_select_bsd` function in “`tpf_select_bsd`–Indicates Read, Write, and Exception Status” on page 629.

### Related Information

- “`FD_COPY`–Copy the File Descriptor Set” on page 135
- “`FD_ISSET`–Return a Value for the File Descriptor in the File Descriptor Set” on page 136
- “`FD_SET`–Add a File Descriptor to a File Descriptor Set” on page 139
- “`FD_ZERO`–Initialize the File Descriptor Set” on page 140
- “`tpf_select_bsd`–Indicates Read, Write, and Exception Status” on page 629.

---

## FD\_COPY–Copy the File Descriptor Set

This function copies the file descriptor set.

### Format

```
#include <sys/time.h>
void FD_COPY(const fd_set* fdset1, fd_set* fdset2)
```

#### **fdset1**

The file descriptor set that is being copied.

#### **fdset2**

The file descriptor set into which the copy is placed.

### Normal Return

Not applicable.

### Error Return

Not applicable.

### Programming Considerations

- If you want to monitor more than 256 file descriptors on a single `tpf_select_bsd` call, the application must define its own `FD_SETSIZE` before the include of header `sys/time.h`.
- The purpose of the FD functions is to set up the input for and check the output from a `tpf_select_bsd` call. The `FD_CLR`, `FD_COPY`, `FD_SET`, and `FD_ZERO` functions set up the input for the `tpf_select_bsd` call. The `FD_ISSET` function checks the output from the `tpf_select_bsd` call.

### Examples

See the example provided for the `tpf_select_bsd` function in “`tpf_select_bsd`–Indicates Read, Write, and Exception Status” on page 629.

### Related Information

- “`FD_CLR`–Remove File Descriptor from the File Descriptor Set” on page 134
- “`FD_ISSET`–Return a Value for the File Descriptor in the File Descriptor Set” on page 136
- “`FD_SET`–Add a File Descriptor to a File Descriptor Set” on page 139
- “`FD_ZERO`–Initialize the File Descriptor Set” on page 140
- “`tpf_select_bsd`–Indicates Read, Write, and Exception Status” on page 629.

---

## FD\_ISSET–Return a Value for the File Descriptor in the File Descriptor Set

This function returns a value for the file descriptor in the file descriptor set.

### Format

```
#include <sys/time.h>
int FD_ISSET(int fd, fd_set* fdset)
```

**fd** The file descriptor.

**fdset**

The file descriptor set.

### Normal Return

Returns a non-zero value if the file descriptor is set in the file descriptor set pointed to by **fdset**; otherwise returns 0.

### Error Return

Not applicable.

### Programming Considerations

- If you want to monitor more than 256 file descriptors on a single `tpf_select_bsd` call, the application must define its own `FD_SETSIZE` before the include of header `sys/time.h`.
- The purpose of the FD functions is to set up the input for and check the output from a `tpf_select_bsd` call. The `FD_CLR`, `FD_COPY`, `FD_SET`, and `FD_ZERO` functions set up the input for the `tpf_select_bsd` call. The `FD_ISSET` function checks the output from the `tpf_select_bsd` call.

### Examples

See the example provided for the `tpf_select_bsd` function in “`tpf_select_bsd`–Indicates Read, Write, and Exception Status” on page 629.

### Related Information

- “`FD_CLR`–Remove File Descriptor from the File Descriptor Set” on page 134
- “`FD_COPY`–Copy the File Descriptor Set” on page 135
- “`FD_SET`–Add a File Descriptor to a File Descriptor Set” on page 139
- “`FD_ZERO`–Initialize the File Descriptor Set” on page 140
- “`tpf_select_bsd`–Indicates Read, Write, and Exception Status” on page 629.

## fdopen—Associate a Stream with an Open File Descriptor

This function associates a stream with an open file descriptor.

### Format

```
#define _POSIX_SOURCE
#include <stdio.h>
FILE *fdopen(int fildes, const char *mode);
```

#### **fildes**

The open file descriptor on which to open a stream.

#### **mode**

The access mode for the stream.

This function associates a stream with an open file descriptor. A stream is a pointer to a FILE structure that contains information about a file. A stream permits user-controlled buffering and formatted input and output.

The specified **mode** must be permitted by the current mode of the file descriptor. For example, if the file descriptor is open-read-only (O\_RDONLY), the corresponding stream cannot be opened for writing (w), for appending (a), or for update (+).

#### **Mode Description**

<b>r</b>	Open for reading.
<b>w</b>	Open for writing.
<b>a</b>	Open for appending.
<b>r+</b>	Open for update (reading and writing).
<b>w+</b>	Open for update (reading and writing).
<b>a+</b>	Open for update at end-of-file (reading and writing).

All of these modes have the same behavior as the corresponding fopen **modes**, except that w and w+ do not truncate the file.

The file position indicator of the new stream is the file offset associated with the file descriptor. The error indicator and end-of-file indicator for the stream are cleared.

### Normal Return

If successful, the fdopen function returns a FILE pointer to the control block for the new stream.

### Error Return

If unsuccessful, the fdopen function returns NULL and sets errno to one of the following:

<b>EINVAL</b>	The specified mode is incorrect or does not match the mode of the open file descriptor.
<b>EBADF</b>	<b>fildes</b> is not a valid open file descriptor.

### Programming Considerations

None.

## fdopen

### Examples

The following example associates **stream** with file descriptor **fd**, which is open for the file `fdopen.file`. The association is made in write mode.

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

main() {
    char fn[]="fdopen.file";
    FILE *stream;
    int fd;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        if ((stream = fdopen(fd, "w")) == NULL) {
            perror("fdopen() error");
            close(fd);
        }
        else {
            fputs("This is a test", stream);
            fclose(stream);
        }
        unlink(fn);
    }
}
```

### Related Information

- “fileno—Get the File Descriptor from an Open Stream” on page 156
- “fopen—Open a File” on page 199
- “open—Open a File” on page 380.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



---

## FD\_SET–Add a File Descriptor to a File Descriptor Set

This function adds a file descriptor to a file descriptor set.

### Format

```
#include <sys/time.h>
void FD_SET(int fd, fd_set* fdset)
```

**fd** The file descriptor.

**fdset**

The file descriptor set.

### Normal Return

Void.

### Error Return

Void.

### Programming Considerations

- If you want to monitor more than 256 file descriptors on a single `tpf_select_bsd` call, the application must define its own `FD_SETSIZE` before the include of header `sys/time.h`.
- The purpose of the FD functions is to set up the input for and check the output from a `tpf_select_bsd` call. The `FD_CLR`, `FD_COPY`, `FD_SET`, and `FD_ZERO` functions set up the input for the `tpf_select_bsd` call. The `FD_ISSET` function checks the output from the `tpf_select_bsd` call.

### Examples

See the example provided for the `tpf_select_bsd` function in “`tpf_select_bsd`–Indicates Read, Write, and Exception Status” on page 629.

### Related Information

- “`FD_CLR`–Remove File Descriptor from the File Descriptor Set” on page 134
- “`FD_COPY`–Copy the File Descriptor Set” on page 135
- “`FD_ISSET`–Return a Value for the File Descriptor in the File Descriptor Set” on page 136
- “`FD_ZERO`–Initialize the File Descriptor Set” on page 140
- “`tpf_select_bsd`–Indicates Read, Write, and Exception Status” on page 629.

---

## **FD\_ZERO–Initialize the File Descriptor Set**

This function initializes the file descriptor set to contain no file descriptors.

### **Format**

```
#include <sys/time.h>
void FD_ZERO(fd_set* fdset)
```

**fdset**

The file descriptor set.

### **Normal Return**

Void.

### **Error Return**

Void.

### **Programming Considerations**

- If you want to monitor more than 256 file descriptors on a single `tpf_select_bsd` call, the application must define its own `FD_SETSIZE` before the include of header `sys/time.h`.
- The purpose of the FD functions is to set up the input for and check the output from a `tpf_select_bsd` call. The `FD_CLR`, `FD_COPY`, `FD_SET`, and `FD_ZERO` functions set up the input for the `tpf_select_bsd` call. The `FD_ISSET` function checks the output from the `tpf_select_bsd` call.

### **Examples**

See the example provided for the `tpf_select_bsd` function in “`tpf_select_bsd`–Indicates Read, Write, and Exception Status” on page 629.

### **Related Information**

- “`FD_CLR`–Remove File Descriptor from the File Descriptor Set” on page 134
- “`FD_COPY`–Copy the File Descriptor Set” on page 135
- “`FD_ISSET`–Return a Value for the File Descriptor in the File Descriptor Set” on page 136
- “`FD_SET`–Add a File Descriptor to a File Descriptor Set” on page 139
- “`tpf_select_bsd`–Indicates Read, Write, and Exception Status” on page 629.

## feof—Test End-of-File Indicator

This function tests the end-of-file (EOF) indicator.

### Format

```
#include <stdio.h>
int feof(FILE *stream);
```

#### **stream**

The stream to be tested.

This function indicates whether the EOF flag is set for the given stream pointed to by **stream**.

The EOF flag is set when a user attempts to read past the EOF flag. Therefore, a read of the last character in the file does not turn on the flag. A subsequent read attempt reaches the EOF flag.

If the file has a simultaneous writer that extends the file, the flag can be turned on by the reader before the file is extended. After the extension becomes visible to the reader, a subsequent read will get the new data and set the flag appropriately. For example, if the read does not read past the EOF flag, the flag is turned off. If a file does not have a simultaneous writer that is extending the file, it is not possible to read past the EOF flag.

A successful repositioning in a file (with `fsetpos`, `rewind`, `fseek`) or a call to the `clearerr` function resets the EOF flag.

### Normal Return

The `feof` function returns a nonzero value if (and only if) the EOF flag is set for **stream**; otherwise, 0 is returned.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example scans the input stream until it reads an EOF character.

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

main() {

    FILE *stream;
    int rc;
    stream = fopen("myfile.dat", "r");

    /* MYFILE.DAT contains 3 characters "abc" */
    while (1) {
        rc = fgetc(stream);
        if (rc == EOF) {
            if (feof(stream)) {
                printf("at EOF\n");
                break;
            }
        }
    }
}
```

## feof

```
        else {  
            printf("error\n");  
            break;  
        }  
    }  
    else  
        printf("read %c\n",rc);  
}  
}
```

### Output

```
read a  
read b  
read c  
at EOF
```

## Related Information

- “clearerr–Reset Error and End-of-File” on page 42
- “fseek–Change File Position” on page 226
- “fsetpos–Set File Position” on page 228
- “rewind–Set File Position to Beginning of File” on page 431.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## ferror—Test for Read/Write Errors

This function tests for read and write errors.

### Format

```
#include <stdio.h>
int ferror(FILE *stream);
```

#### **stream**

The stream to be tested.

This function tests for an error when reading from, or writing to, the specified **stream**. If an error occurs, the error indicator for **stream** remains set until you close the stream, call the rewind function, or call the `clearerr` function.

If an incorrect parameter is detected during an input/output (I/O) function call, the error flag is not turned on.

### Normal Return

The `ferror` function returns a nonzero value to indicate an error for the stream pointed to by **stream**; otherwise, it returns 0.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example puts data out to a stream and then checks that a write error has not occurred.

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    char *string = "Important information";
    stream = fopen("myfile.dat", "w");

    fprintf(stream, "%s\n", string);
    if (ferror(stream))
    {
        printf("write error\n");
        clearerr(stream);
    }
    if (fclose(stream))
        printf("fclose error\n");
}
```

### Related Information

- “`clearerr`—Reset Error and End-of-File” on page 42
- “`rewind`—Set File Position to Beginning of File” on page 431.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## fflush–Write Buffer to File

This function writes buffer to a file.

### Format

```
#include <stdio.h>
int fflush(FILE *stream);
```

**stream**

The stream to be flushed.

This function flushes any buffered data written to the stream pointed to by **stream**. If **stream** is NULL, it flushes all streams that are open for writing, and all streams that are open for updating that have data buffered for output. There is no effect on streams that are opened for reading or updating if the last operation on the stream was not a write.

**Note:** The TPF system automatically flushes buffers when you close the stream or when a program ends normally without closing the stream.

The buffering mode and the file type can have an effect on when output data is flushed.

The stream remains open after the fflush function call. If **stream** is open for update, a read operation can immediately follow the fflush function call.

### Normal Return

If successful, the fflush function returns a value 0.

### Error Return

The fflush function returns EOF if an error occurs. When flushing all open files, a failure to flush any of the files causes EOF to be returned. However, flushing will continue on any other open files that can be flushed successfully.

### Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

### Examples

The following example flushes a stream buffer and tests for the returned value of 0 to see if the flushing was successful.

```
#include <stdio.h>

int retval;
int main(void)
{
    FILE *stream;
    stream = fopen("myfile.dat", "w");

    retval=fflush(stream);
    printf("return value=%i",retval);
}
```

## Related Information

- “setbuf–Control Buffering” on page 460
- “setvbuf–Control Buffering” on page 474
- “ungetc–Push Character to Input Stream” on page 664.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## fgetc—Read a Character

This function reads a character.

### Format

```
#include <stdio.h>
int fgetc(FILE *stream);
```

#### **stream**

The stream to be read.

This function reads a single-byte unsigned character from the input stream pointed to by **stream** at the current position and increases the associated file pointer so that it points to the next character.

The `getc` function has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a read and a subsequent write, there must be an intervening reposition unless an end-of-file (EOF) has been reached.

### Normal Return

If successful, the `fgetc` function returns the character that was read as an integer.

### Error Return

An EOF return value indicates an error or an EOF condition. Use the `feof` or `ferror` function to determine if the EOF value indicates an error or the end of the file. Note that EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does **not** turn on the EOF indicator.

### Programming Considerations

None.

### Examples

The following example gathers a line of input from a stream and tests to see if the file can be opened. If the file cannot be opened, the function `perror` is called.

```
#include <stdio.h>
#define MAX_LEN 80

int main(void)
{
    FILE *stream;
    char buffer[MAX_LEN + 1];
    int i, ch;

    if ((stream = fopen("myfile.dat", "r")) != NULL) {
        for (i = 0; (i < (sizeof(buffer)-1) &&
            ((ch = fgetc(stream)) != EOF) && (ch != '\n')); i++)
            printf("character is %d\n", ch);
        buffer[i] = ch;

        buffer[i] = '\0';

        if (fclose(stream))
            perror("fclose error");
    }
```



```
    }  
    else  
        perror("fopen error");  
}
```

## Related Information

- “feof—Test End-of-File Indicator” on page 141
- “ferror—Test for Read/Write Errors” on page 143
- “fputc—Write a Character” on page 209
- “getc, getchar—Read a Character from Input Stream” on page 246.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## fgetpos—Get File Position

This function gets a file position.

### Format

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

#### **stream**

The stream whose current position will be obtained.

#### **pos**

The address of an object of type `fpos_t` into which the current file location will be stored.

This function stores the current value of the file pointer associated with **stream** into the object pointed to by **pos**. The value pointed to by **pos** can be used later in a call to the `fsetpos` function to reposition the stream pointed to by **stream**.

Both the `fgetpos` and `fsetpos` functions also save state information for wide-oriented files. The value stored in **pos** is not specified and is used only by the `fsetpos` function.

The position returned by the `fgetpos` function is affected by the `ungetc` functions. Each call to these functions causes the file position indicator to be backed up from the position where the `ungetc` function was issued. For details about how `ungetc` affects `fgetpos` function behavior, see “`ungetc`—Push Character to Input Stream” on page 664.

### Normal Return

If successful, the `fgetpos` function returns 0.

### Error Return

If unsuccessful, the `fgetpos` function returns a nonzero value.

### Programming Considerations

None.

### Examples

The following example opens the `myfile.dat` file for reading. The current file pointer position is stored into the **pos** variable.

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    int retcode;
    fpos_t pos;

    stream = fopen("myfile.dat", "rb");

    /* The value returned by fgetpos can be used by fsetpos()
       to set the file pointer if 'retcode' is 0 */
    retcode = fgetpos(stream, &pos);
}
```

```
if ((retcode = fgetpos(stream, &pos)) == 0)
    printf("Current position of file pointer found\n");
fclose(stream);
}
```

## Related Information

- “fseek—Change File Position” on page 226
- “fsetpos—Set File Position” on page 228
- “ftell—Get Current File Position” on page 234
- “ungetc—Push Character to Input Stream” on page 664.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## fgets—Read a String from a Stream

This function reads a string from a stream.

### Format

```
#include <stdio.h>
char *fgets(char *string, int n, FILE *stream);
```

#### **string**

The address of the first character in the buffer into which the string will be read.

**n** The size of the string buffer.

#### **stream**

The stream to be read.

This function reads bytes from a stream pointed to by **stream** into an array pointed to by **string**, starting at the position indicated by the file position indicator. Reading continues until the number of characters read is equal to **n**−1, or until a new-line character (`\n`), or until the end of the stream, whichever comes first. The `fgets` function stores the result in **string** and adds a null character (`\0`) to the end of the string. The **string** includes the new-line character, if read.

The `fgets` function has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an EOF has been reached.

### Normal Return

If successful, the `fgets` function returns a pointer to the **string** buffer.

### Error Return

If unsuccessful, the `fgets` function returns a NULL pointer.

If **n** is less than or equal to 0, the `fgets` function indicates a domain error and `errno` is set to `EDOM` to indicate the cause of the failure.

If **n** equals 1 (which indicates a valid result), the string buffer has room only for the null terminator; nothing is physically read from the file. (Such an operation is still considered a read operation, so it cannot immediately follow a write operation unless there is an intervening flush or reposition operation first.)

If **n** is greater than 1, the `fgets` function will only fail if an I/O error occurs or if end-of file (EOF) is reached, and no data is read from the file.

The `ferror` and `feof` functions are used to distinguish between a read error and an EOF.

**Note:** EOF is reached only when an attempt is made to read *past* the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

If EOF is reached after data has already been read into the string buffer, the fgets function returns a pointer to the string buffer to indicate success. A subsequent call would result in NULL being returned because EOF would be reached without any data being read.

## Programming Considerations

None.

## Examples

The following example gets a line of input from a data stream. It reads no more than MAX\_LEN - 1 characters or up to a new-line character from the stream.

```
#include <stdio.h>
#define MAX_LEN 100

int main(void)
{
    FILE *stream;
    char line[MAX_LEN], *result;

    stream = fopen("myfile.dat", "r");

    if ((result = fgets(line, MAX_LEN, stream)) != NULL)
        printf("The string is %s\n", result);

    if (fclose(stream))
        printf("fclose error\n");
}
```

## Related Information

- “feof—Test End-of-File Indicator” on page 141
- “ferror—Test for Read/Write Errors” on page 143
- “fgetc—Read a Character” on page 146
- “fputs—Write a String” on page 211
- “gets—Obtain Input String” on page 273
- “puts—Put String to Standard Output Stream” on page 402.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## filec—File a Record: Basic

This function writes a working storage block from an entry control block (ECB) data **level** to file. The ECB must be holding a storage block on the specified level.

This service files a record to either VFA or DASD.

The `filec` function returns the block of storage to the appropriate pool that is referred to in the CBRW at the specified level.

## Format

```
#include <tpfio.h>
void filec(enum t_lvl level);
```

### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this core block reference word (CBRW) level is the record to be filed.

## Normal Return

Void.

## Error Return

Not applicable.

## Programming Considerations

- The `filec` function cannot be issued on a record that is part of the suspended commit scope of the ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filec()` record X
  3. `tx_suspend_tpf()`
  4. `filec()` record X.
- The specified data level is initialized to indicate that a block of storage is no longer held. Specifying an invalid data level results in a system error with exit.
- The FARW at the specified level is unchanged.
- The program identification is placed in the record header.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level, and if the file address contained at the specified level is valid. In addition, the record type at the specified level is checked with the record type in the record. If any condition is violated, control is transferred to the system error routine. When the record code check in the FARW is zero, the control program ignores verification of the record code check. If the record code check is nonzero, TPF verifies that the code specified in the FARW is the same as the code in the header of the record. If the codes are not equal, control is transferred to the system error routine.
- The block of storage containing the data to be stored is no longer available to the operational program. The operational program can use the specified CBRW immediately on return from the function.
- The status of the operation can never be determined by the operational program.
- TPF transaction services processing affects `filec` processing in the following ways:

- Hardening to the DASD surface will occur only at commit time.
- When rollback occurs, filec changes are discarded and hardening does not occur. The DASD surface remains unchanged.
- If a system error occurs because of an ID or record code check (RCC), processing ends as if a rollback was issued.

## Examples

The following example writes the data in the working storage block on level D4 to file and releases the block.

```
#include <tpfio.h>
⋮
filec(D4);
```

## Related Information

- “filec\_ext–File a Record with Extended Options: Basic” on page 154
- “file\_record–File a Record: Higher Level” on page 158
- “filnc–File a Record with No Release” on page 165
- “filuc–File and Unhold a Record” on page 170.

## filec\_ext—File a Record with Extended Options: Basic

This function writes a working storage block from an entry control block (ECB) data level to file. The ECB must be holding a storage block on the specified level.

This service files a record to either VFA or DASD.

The `filec_ext` function returns the block of storage to the appropriate pool that is referred to in the CBRW at the specified level.

### Format

```
#include <tpfio.h>
void filec_ext(enum t_lvl level, unsigned int ext);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this core block reference word (CBRW) level is the record to be filed.

#### ext

Sum of the following bit flags that are defined in `tpfio.h`.

#### FILE\_GDS

Use `FILE_GDS` to specify that the record to be filed resides in a general file or general data set. If `FILE_GDS` is not specified, `filec_ext` accesses the record on the online database.

#### FILE\_NOTAG

The TPF system code that places the program identification in the record header is bypassed. This flag should only be used when the application updating the record has placed the required program identification in the header directly.

**Note:** If neither of the above flags are needed, the default extended options flag, `FILE_DEFEXT`, should be coded. Consider using the `filec` function.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- The `filec` function cannot be issued on a record that is part of the suspended commit scope of the ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filec_ext()` record X
  3. `tx_suspend_tpf()`
  4. `filec_ext()` record X.
- The specified data level is initialized to indicate that a block of storage is no longer held. Specifying an invalid data level results in a system error with `exit`.
- The FARW at the specified level is unchanged.



- If you specify FILE\_NOTAG, the TPF system bypasses the system code that updates the record header with the program identification. If you omit FILE\_NOTAG, the TPF system places the program identification in the record header.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level, and if the file address contained at the specified level is valid. In addition, the record type at the specified level is checked with the record type in the record. If any condition is violated, control is transferred to the system error routine. When the record code check (RCC) in the FARW is zero, the control program ignores verification of the record code check. If the record code check is nonzero, TPF verifies that the code specified in the FARW is the same as the code in the header of the record. If the codes are not equal, control is transferred to the system error routine.
- The block of storage containing the data to be stored is no longer available to the operational program. The operational program can use the specified CBRW immediately on return from the function.
- The status of the operation can never be determined by the operational program.
- TPF transaction services processing affects filec\_ext processing in the following ways:
  - Hardening to the DASD surface will only occur at commit time.
  - When rollback occurs, filec\_ext changes are discarded and hardening does not occur. The DASD surface remains unchanged.
  - If a system error occurs because of an ID or record code check, processing ends as if a rollback was issued.
  - Files to general files or general data sets are not considered part of the commit scope and are not affected by commit scope processing.

## Examples

The following example writes the data in the working storage block on level D7 to a general data set, bypasses the record header update, and releases the block.

```
#include <tpfio.h>
:
filec_ext(D7,FILE_NOTAG|FILE_GDS);
```

## Related Information

- “filec—File a Record: Basic” on page 152
- “file\_record\_ext—File a Record with Extended Options: Higher Level” on page 161
- “filnc\_ext—File a Record with No Release and Extended Options” on page 167
- “filuc\_ext—File and Unhold a Record with Extended Options” on page 172.

## fileno—Get the File Descriptor from an Open Stream

This function returns the file descriptor number associated with a specified stream.

### Format

```
#define _POSIX_SOURCE
#include <stdio.h>
int fileno(const FILE *stream);
```

#### **stream**

The stream for which the associated file descriptor will be returned.

The argument **stream** points to a FILE structure controlling a stream.

The unistd.h header file defines the following macros, which are constants that map to the file descriptors of the standard streams:

#### **STDIN\_FILENO**

Standard input, stdin (value 0).

#### **STDOUT\_FILENO**

Standard output, stdout (value 1).

#### **STDERR\_FILENO**

Standard error, stderr (value 2).

**Note:** stdin, stdout, and stderr are macros, not constants.

### Normal Return

If successful, the fileno function returns the file descriptor number associated with an open stream (that is, one opened with the fopen, fdopen, or freopen functions).

### Error Return

If unsuccessful, fileno returns -1 and sets errno to EBADF, which indicates one of the following:

- **stream** points to a closed stream.
- **stream** is an incorrect stream pointer.

### Programming Considerations

None.

### Examples

The following example shows one use of the fileno function.

```
#define _POSIX_SOURCE
#include <errno.h>
#include <stdio.h>

main() {
    FILE *stream;
    char my_file[]="my.file";

    printf("fileno(stdin) = %d\n", fileno(stdin));

    if ((stream = fopen(my_file, "w")) == NULL)
        perror("fopen() error");
    else {
        printf("fileno() of the file is %d\n", fileno(stream));
    }
}
```

```
        fclose(stream);  
        remove(my_file);  
    }  
}
```

**Output**

```
fileno(stdin) = 0  
fileno() of the file is 3
```

**Related Information**

- “fopen—Open a File” on page 199
- “freopen—Redirect an Open File” on page 215
- “fdopen—Associate a Stream with an Open File Descriptor” on page 137
- “open—Open a File” on page 380.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## file\_record–File a Record: Higher Level

This service files a record to either VFA or DASD.

### Format

```
#include <tpfio.h>
void file_record(enum t_lvl level, const unsigned int *address,
                 const char *id, unsigned char rcc, enum t_act type);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this core block reference word (CBRW) level is the record to be filed.

#### address

A pointer to the file address indicating where the record will be filed.

**id** A pointer to the 2-character record ID.

#### rcc

An unsigned character that represents the value to be assigned to the record code check field of the record being filed.

#### type

The disposition of the record's hold status, or the associated working block. This argument must belong to the enumeration type `t_act`, defined in `tpfio.h`.

#### NOHOLD

The record is not in hold status and the associated working storage block can be returned to the system.

#### UNHOLD

The record is in hold status and the associated working storage block can be returned to the system.

#### NOREL

The record is not in hold status and the associated working storage block should not be returned to the system.

### Normal Return

Void. The associated working storage buffer block has been made unavailable to the operational program. For **NOREL** calls, the buffer block will be available to the operational program following a subsequent `waitc` call.

### Error Return

Not applicable.

### Programming Considerations

- The `file_record` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `file_record()` record X
  3. `tx_suspend_tpf()`
  4. `file_record()` record X.

- On **NOHOLD** and **UNHOLD** calls the status of the operation can never be determined. You must code a `waitc` call to determine the status of the file action on an **NOREL** request.
- The CBRW of the data level named as the first argument must be occupied by a working storage block of the same size as the file record. This block will be considered the image of the record to be filed.
- The FARW of the data level named as the first argument is changed by this function with the **address**, **id**, and **rcc** arguments. If either the **address** or **id** arguments are coded with the defined term **NULL**, initialization of the indicated FARW with the null item is not performed. If the defined term **RECID\_RESET** is specified for the **id** argument, the record id in the FARW will be set to binary zeros.
- For **UNHOLD** requests, the file address must have been held by the issuing ECB. This address is removed from the record hold table following I/O completion.
- For **NOREL** and **NOHOLD** requests, if the file address is present in the record hold table, it will remain there through this call.
- This function may be used for general file output, provided that the file address has been calculated by use of the `gdsnc`, `gdsrsc`, or `raisr` functions. General file users should be aware that argument **address** should be coded as **NULL**, as the file address is placed on the FARW level by all of the previous functions.
- Control is transferred to the system error routine if the **id** and **rcc** values specified do not match the corresponding fields in the record image. Coding the **rcc** as `\0` causes this check to be bypassed.
- TPF transaction services processing affects `file_record` processing in the following ways:
  - Hardening to the DASD surface will only occur at commit time.
  - When rollback occurs, `filec` changes are discarded and hardening does not occur. The DASD surface remains unchanged.
  - If a system error occurs because of an ID or record code check (RCC), processing ends as if a rollback was issued.
  - `waitc` processing does not report hardware errors for **NOREL** when the `waitc` function is issued in the commit scope.
  - When the file address returns from the **NOREL** file, it appears to be unheld from the point of view of the program. For requests from outside the commit scope, the file address still appears to be held. When the commit has completed successfully, the file address will be unheld and any waiting requests will be serviced.

## Examples

The following example files a record from D6 with **id** set to IM, the **rcc** set to zero, and the file address taken from the forward chain field of another record.

```
#include <tpfio.h>
struct im0im
{
    char            im0bid[2];           /* record ID           */
    unsigned char    im0rcc;             /* record code check   */
    unsigned char    im0ctl;             /* control byte        */
    char            im0pgm[4];           /* program stamp       */
    unsigned long int im0fch;             /* forward chain       */
    unsigned long int im0bch;             /* backward chain      */
    short int        im0cct;             /* byte count          */
    .
    (data fields)
}
```

## file\_record

```
        .  
    } *inm;  
    .  
    .  
file_record(D6,(const unsigned int *)&(inm->im0fch),"IM",'\0',NOHOLD);
```

## Related Information

- “file\_record\_ext–File a Record with Extended Options: Higher Level” on page 161
- “find\_record–Find a Record” on page 178
- “unfrc–Unhold a File Record” on page 661
- “waitc–Wait For Outstanding I/O Completion” on page 686
- “findc–Find a Record” on page 174.

## file\_record\_ext—File a Record with Extended Options: Higher Level

This service files a record to either virtual file access (VFA) or DASD.

### Format

```
#include <tpfio.h>
void file_record_ext(enum t_lvl level, const unsigned int *address,
                    const char *id, unsigned char rcc,
                    enum t_act type, unsigned int ext);
```

or

```
#include <tpfio.h>
void file_record_ext(TPF_DECB *decb, TPF_FA8 *fa8,
                    const char *id, unsigned char rcc,
                    enum t_act type, unsigned int ext);
```

#### level

One of 16 possible values representing a valid entry control block (ECB) data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this core block reference word (CBRW) level is the record to be filed.

#### decb

A pointer to a data event control block (DECB). The working storage block anchored off the CBRW of the DECB is the record to be filed.

#### address

A pointer to the file address indicating where the record will be filed.

#### fa8

A pointer to an 8-byte file address indicating where the record will be filed.

**id** A pointer to a 2-character record ID string that must match the ID characters in the record to be filed. This check can be bypassed by coding the term **RECID\_RESET**, defined in `tpfio.h`.

#### rcc

An unsigned character that represents the value to be assigned to the record code check (RCC) field of the record being filed.

#### type

The disposition of the record's hold status, or the associated working block. This parameter must belong to the enumeration type `t_act`, defined in `tpfio.h`.

#### NOHOLD

The record is not in hold status and the associated working storage block can be returned to the system.

#### UNHOLD

The record is in hold status and the associated working storage block can be returned to the system.

#### NOREL

The record is not in hold status and the associated working storage block should not be returned to the system.

#### ext

Logical or (|) of the following bit flags that are defined in `tpfio.h`.

## file\_record\_ext

### FILE\_GDS

Use FILE\_GDS to specify that the record to be filed resides in a general file or general data set. If FILE\_GDS is not specified, file\_record\_ext accesses the record on the online database.

### FILE\_NOTAG

The TPF system code that places the program identification in the record header is bypassed. This flag should only be used when the application updating the record has placed the required program identification in the header directly.

**Note:** If flags FILE\_GDS or FILE\_NOTAG are not needed, code the default extended options flag, FILE\_DEFEXT. Consider using the file\_record function.

## Normal Return

Void. The associated working storage buffer block has been made unavailable to the operational program. For **NOREL** calls, the buffer block will be available to the operational program following a subsequent waitc call.

## Error Return

Not applicable.

## Programming Considerations

- The file\_record\_ext function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. tx\_begin()
  2. file\_record\_ext() record X
  3. tx\_suspend\_tpf()
  4. file\_record\_ext() record X.
- On **NOHOLD** and **UNHOLD** calls the status of the operation can never be determined. The programmer must code a waitc call to determine the status of the file action on an **NOREL** request.
- The CBRW of the ECB data level or DECB named as the first parameter must be occupied by a working storage block of the same size as the file record. This block is considered the image of the record to be filed.
- The FARW of the ECB data level or DECB named as the first parameter is changed by this function with the **address**, **fa8**, **id**, and **rcc** parameters. If either the **address**, **fa8**, or **id** parameters are coded with the defined term **NULL**, initialization of the indicated FARW with the null item is not performed. If the defined term **RECID\_RESET** is specified for the **id** parameter, the record id in the FARW will be set to binary zeros.
- For **UNHOLD** requests, the file address must have been held by the issuing ECB. This address is removed from the record hold table following I/O completion.
- For **NOREL** and **NOHOLD** requests, if the file address is present in the record hold table, it will remain there through this call.
- This function may be used for general file output, provided that the file address has been calculated by using the gdsnc, gdsrc, or raisa functions. General file



users should be aware that the **address** or **fa8** parameter must be coded as **NULL**, as the file address is placed on the FARW level or DECB by all of the previous functions.

- Control is transferred to the system error routine if the **id** and **rcc** values specified do not match the corresponding fields in the record image. Coding the **rcc** as `\0` causes this check to be bypassed.
- If you specify **FILE\_NOTAG**, the TPF system bypasses the system code that updates the record header with the program identification. If you omit **FILE\_NOTAG**, the TPF system places the program identification in the record header.
- TPF transaction services processing affects `file_record_ext` processing in the following ways:
  - Hardening to the DASD surface will only occur at commit time.
  - When rollback occurs, `file_record_ext` changes are discarded and hardening does not occur. The DASD surface remains unchanged.
  - If a system error occurs because of an ID or record code check, processing ends as if a rollback was issued.
  - `waitc` processing does not report hardware errors for **NOREL** when the `waitc` function is issued within the commit scope.
  - When the file address returns from the **NOREL** file, it appears to be unheld from the point of view of the program. For requests from outside the commit scope, the file address still appears to be held. When the commit completes, the file address will be unheld and any waiting requests will be serviced.
  - Files to general files or general data sets are not considered part of the commit scope and are not affected by commit scope processing.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example writes the data in the working storage block on level D7 to a general data set, bypasses the record header update, and releases the block. The **id** is CD, the **rcc** is zero, and the file address is taken from the forward chain field of another record. On return, the file copy of the record is available to other ECBs.

```

:
file_record_ext(D7,(const unsigned int *)&(inm->im0fch),CD,'\0',
               UNHOLD,FILE_NOTAG|FILE_GDS);

```

The following example writes the data in the working storage block on a DECB and unholds the record. The file **address** and record **id** are specified in the DECB.

```

#include <tpfio.h>

TPF_DECB *decb
:
file_record_ext(decb, NULL, NULL, '\0', UNHOLD, FILE_NOTAG);

```

## Related Information

- “file\_record–File a Record: Higher Level” on page 158
- “find\_record\_ext–Find a Record with Extended Options” on page 181
- “unfrc\_ext–Unhold a File Record with Extended Options” on page 662

## **file\_record\_ext**

- “waitc—Wait For Outstanding I/O Completion” on page 686
- “findc\_ext—Find a Record with Extended Options” on page 176.

See *TPF Application Programming* for more information about DECBs.

## filnc—File a Record with No Release

This function writes a working storage block from an entry control block (ECB) data level to file. The ECB must be holding a storage block on the specified level.

This service files a record to either VFA or DASD.

The `filnc` function does not return the block of storage to the appropriate pool but retains it on the specified level following return from a subsequent `waitc`.

### Format

```
#include <tpfio.h>
void      filnc(enum t_lvl level);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this CBRW level is the record to be filed.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The `filnc` function cannot be issued on a record that is part of an ECB's suspended commit scope. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filnc()` record X
  3. `tx_suspend_tpf()`
  4. `filnc()` record X
- The FARW at the specified level is unchanged.
- The program identification is placed in the record header.
- Specifying an invalid data level results in a system error with `exit`.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level and if the file address contained at the specified level is valid. In addition, the record type at the specified level is checked with the record type in the record. If any condition is violated, control is transferred to the system error routine. When the record code check in the FARW is zero, the control program ignores verification of the record code check. If the record code check is nonzero, the TPF system verifies that the code specified in the FARW is the same as the code in the header of the record. If the codes are not equal, control is transferred to the system error routine.
- To ensure completion of the operation, `waitc` must be executed.
- An error is posted during the execution of `waitc` only if the prime duplicate copy of the record were not successfully updated.
- TPF transaction services processing affects `filnc` processing in the following ways:
  - Hardening to the DASD surface will only occur at commit time.

## filnc

- When a group of updated records is hardened to DASD, the filing sequence originally specified by the application program is maintained. Therefore, applications that rely on a specific sequence of `filnc` and `waitc` function calls to control the order of updates in a chain of related records will not lose this ability when placed in a commit scope.

**Note:** This guarantee of filing sequence only applies to file requests that are part of the commit scope. It does **not** include filing to general files or general data sets.

- When rollback occurs, `filnc` changes are discarded and hardening does not occur. The DASD surface remains unchanged.
- If a system error occurs because of an ID or record code check, processing ends as if a rollback was issued.
- `waitc` processing does not report hardware errors when the `waitc` function is issued in a commit scope.

## Examples

The following example writes the data in the working storage block on level DF to file. The working storage block remains attached to the data level.

```
#include <tpfio.h>
:
:
filnc(DF);
if (waitc())
{
    serrc_op(SERRC_EXIT,0x12345,"I/O ERROR OCCURRED",NULL);
}
/* This serrc_op call is the ISO-C version of the TARGET(TPF) call:
   errno = 0x1234;
   perror("I/O ERROR OCCURRED");
   abort();
*/
```

## Related Information

- “`filnc_ext`—File a Record with No Release and Extended Options” on page 167
- “`file_record`—File a Record: Higher Level” on page 158
- “`filec`—File a Record: Basic” on page 152
- “`filuc`—File and Unhold a Record” on page 170
- “`waitc`—Wait For Outstanding I/O Completion” on page 686.

## filnc\_ext—File a Record with No Release and Extended Options

This function writes a working storage block from an entry control block (ECB) data level to file. The ECB must be holding a storage block on the specified level.

This service files a record to either VFA or DASD.

The `filnc_ext` function does not return the block of storage to the appropriate pool but retains it on the specified level following return from a subsequent `waitc`.

### Format

```
#include <tpfio.h>
void      filnc_ext(enum t_lvl level, unsigned int ext);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this CBRW level is the record to be filed.

#### ext

Sum of the following bit flags that are defined in `tpfio.h`.

#### FILE\_GDS

Use `FILE_GDS` to specify that the record to be filed resides in a general file or general data set. If `FILE_GDS` is not specified, `filnc_ext` accesses the record on the online database.

#### FILE\_NOTAG

The TPF system code that places the program identification in the record header is bypassed. This flag should only be used when the application updating the record has placed the required program identification in the header directly.

**Note:** If neither of the above flags are needed the default extended options flag, `FILE_DEFEXT`, should be coded. Consider using the `filnc` function.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- The `filnc_ext` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filnc_ext()` record X
  3. `tx_suspend_tpf()`
  4. `filnc_ext()` record X.
- The FARW at the specified level is unchanged.

## filnc\_ext

- If you specify FILE\_NOTAG, the TPF system bypasses the system code that updates the record header with the program identification. If you omit FILE\_NOTAG, the TPF system places the program identification in the record header.
- Specifying an invalid data level results in a system error with exit.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level and if the file address contained at the specified level is valid. In addition, the record type at the specified level is checked with the record type in the record. If any condition is violated, control is transferred to the system error routine. When the record code check in the FARW is zero, the control program ignores verification of the record code check. If the record code check is nonzero, the TPF system verifies that the code specified in the FARW is the same as the code in the header of the record. If the codes are not equal, control is transferred to the system error routine.
- To ensure completion of the operation, waitc must be executed.
- An error is posted during the execution of waitc only if neither the prime nor duplicate copy of the record was successfully updated.
- TPF transaction services processing affects filnc\_ext processing in the following ways:
  - Hardening to the DASD surface will only occur at commit time.
  - When a group of updated records is hardened to DASD, the filing sequence originally specified by the application program is maintained. Therefore, applications that rely on a specific sequence of filnc and waitc function calls to control the order of updates in a chain of related records will not lose this ability when placed in a commit scope.

**Note:** This guarantee of filing sequence only applies to file requests that are part of the commit scope. It does **not** include filing to general files or general data sets.

- When rollback occurs, filnc\_ext changes are discarded and hardening does not occur. The DASD surface remains unchanged.
- If a system error occurs because of an ID or record code check (RCC), processing ends as if a rollback was issued.
- waitc processing does not report hardware errors when the waitc function is issued in a commit scope.
- Files to general files or general data sets are not considered part of the commit scope and are not affected by commit scope processing.

## Examples

The following example writes the data in the working storage block on level D7 to a general data set and bypasses the record header update. The working storage block remains attached to the data level.

```
#include <tpfio.h>
:
:
filnc_ext(D7,FILE_NOTAG|FILE_GDS);
if (waitc())
{
    serrc_op(SERRC_EXIT,0x12345,"I/O ERROR OCCURRED",NULL);
}
/* This serrc_op call is the ISO-C version of the TARGET(TPF) call:
   errno = 0x1234;
   perror("I/O ERROR OCCURRED");
   abort();
*/
```

## Related Information

- “filnc–File a Record with No Release” on page 165
- “file\_record\_ext–File a Record with Extended Options: Higher Level” on page 161
- “filec\_ext–File a Record with Extended Options: Basic” on page 154
- “filuc\_ext–File and Unhold a Record with Extended Options” on page 172
- “waitc–Wait For Outstanding I/O Completion” on page 686.

## filuc—File and Unhold a Record

This function writes a working storage block from an entry control block (ECB) data level to file and releases it from exclusive control (unholds it) from the requesting ECB. The ECB must be holding a storage block on the specified level.

This service files a record to either VFA or DASD.

The `filuc` function returns the block of storage to the appropriate pool that is referred to in the CBRW at the specified level.

### Format

```
#include <tpfio.h>
void      filuc(enum t_lvl level);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this core block reference word (CBRW) level is the record to be filed.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- The `filuc` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filuc()` record X
  3. `tx_suspend_tpf()`
  4. `filuc()` record X.
- The specified CBRW is initialized to indicate that a block of storage is no longer held. Specifying an invalid data level results in a system error with exit.
- The FARW at the specified level is unchanged, but the file copy of the record is now available to other ECBs requesting exclusive control (hold).
- The program identification is placed in the record header.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level and if the file address contained at the specified level is valid. In addition, the record type at the specified level is checked with the record type in the record. If any condition is violated, control is transferred to the system error routine. When the record code check in the FARW is zero, the control program ignores verification of the record code check. If the record code check is nonzero, the TPF system verifies that the code specified in the FARW is the same as the code in the header of the record. If the codes are not equal, control is transferred to the system error routine.
- The block of storage containing the data to be stored is no longer available to the operational program. The operational program can use the specified CBRW immediately upon return from the function.
- The status of the operation can never be determined by the operational program.



- TPF transaction services processing affects filuc processing in the following ways:
  - Hardening to the DASD surface will only occur at commit time.
  - When rollback occurs, filuc changes are discarded and hardening does not occur. The DASD surface remains unchanged.
  - If a system error occurs because of an ID or record code check (RCC), processing ends as if a rollback was issued.
  - When the file address returns from the filuc call, it appears to be unheld from the point of view of the program. For requests from outside the commit scope, the file address still appears to be held. When the commit has completed successfully, the file address will be unheld and any waiting requests will be serviced.
  - Files to general files or general data sets are not considered part of the commit scope and are not affected by commit scope processing.

**Note:** If a request is waiting to hold the same record, it is serviced by the execution of a storage-to-storage move. This gives the waiting ECB faster access to the record.

## Examples

The following example writes the data in the working storage block on level D4 to file and releases the block. On return, the file copy of the record is available to other ECBs.

```
#include <tpfio.h>
:
:
filuc(D0);
```

## Related Information

- “filuc\_ext–File and Unhold a Record with Extended Options” on page 172
- “file\_record–File a Record: Higher Level” on page 158
- “filec–File a Record: Basic” on page 152
- “filnc–File a Record with No Release” on page 165.

## filuc\_ext–File and Unhold a Record with Extended Options

This function writes a working storage block from an entry control block (ECB) data level to file and releases it from exclusive control (unholds it) from the requesting ECB. The ECB must be holding a storage block on the specified level.

This service files a record to either VFA or DASD.

The `filuc_ext` function returns the block of storage to the appropriate pool that is referred to in the CBRW at the specified level.

### Format

```
#include <tpfio.h>
void      filuc_ext(enum t_lvl level, unsigned int ext);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this core block reference word (CBRW) level is the record to be filed.

#### ext

Sum of the following bit flags, that are defined in `tpfio.h`.

#### FILE\_GDS

Use `FILE_GDS` to specify that the record to be filed and unheld resides in a general file or general data set. If `FILE_GDS` is not specified, `filuc_ext` accesses the record on the online database.

#### FILE\_NOTAG

The TPF system code that places the program identification in the record header is bypassed. This flag should only be used when the application updating the record has placed the required program identification in the header directly.

**Note:** If neither of the above flags are needed the default extended options flag, `FILE_DEFEXT`, should be coded. Consider using the `filuc` function.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- The `filuc_ext` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filuc_ext()` record X
  3. `tx_suspend_tpf()`
  4. `filuc_ext()` record X.
- The specified CBRW is initialized to indicate that a block of storage is no longer held. Specifying an invalid data level results in a system error with exit.

- The FARW at the specified level is unchanged, but the file copy of the record is now available to other ECBs requesting exclusive control (hold).
- If you specify FILE\_NOTAG, the TPF system bypasses the system code that updates the record header with the program identification. If you omit FILE\_NOTAG, the TPF system places the program identification in the record header.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level and if the file address contained at the specified level is valid. In addition, the record type at the specified level is checked with the record type in the record. If any condition is violated, control is transferred to the system error routine. When the record code check in the FARW is zero, the control program ignores verification of the record code check. If the record code check is nonzero, the TPF system verifies that the code specified in the FARW is the same as the code in the header of the record. If the codes are not equal, control is transferred to the system error routine.
- The block of storage containing the data to be stored is no longer available to the operational program. The operational program can use the specified CBRW immediately upon return from the function.
- The status of the operation can never be determined by the operational program.
- TPF transaction services processing affects filuc\_ext processing in the following ways:
  - Hardening to the DASD surface will only occur at commit time.
  - When rollback occurs, filuc\_ext changes are discarded and hardening does not occur. The DASD surface remains unchanged.
  - If a system error occurs because of an ID or record code check (RCC), processing ends as if a rollback was issued.
  - When the file address returns from the filuc\_ext call, it appears to be unheld from the point of view of the program. For requests from outside the commit scope, the file address still appears to be held. When the commit has completed successfully, the file address will be unheld and any waiting requests will be serviced.
  - Files to general files or general data sets are not considered part of the commit scope and are not affected by commit scope processing.

**Note:** If a request is waiting to hold the same record, it is serviced by the execution of a storage-to-storage move. This gives the waiting ECB faster access to the record.

## Examples

The following example writes the data in the working storage block on level D7 to a general data set, bypasses the record header update, and releases the block.

```
#include <tpfio.h>
:
:
filuc_ext(D7,FILE_GDS|FILE_NOTAG);
```

## Related Information

- “filuc–File and Unhold a Record” on page 170
- “file\_record\_ext–File a Record with Extended Options: Higher Level” on page 161
- “filec\_ext–File a Record with Extended Options: Basic” on page 154
- “filnc\_ext–File a Record with No Release and Extended Options” on page 167.

## findc—Find a Record

This function reads a record and attaches the block of working storage containing it to the specified data level of the ECB. The ECB must not be holding a storage block on the specified level.

This service finds a record that resides either in VFA or DASD.

### Format

```
#include <tpfio.h>
void findc(enum t_lvl level);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The data record being retrieved is attached to this level.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The `findc` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `findc()` record X
  3. `tx_suspend_tpf()`
  4. `findc()` record X.
- The specified CBRW is initialized to indicate that a block of storage is attached to the specified data level. Specifying an invalid data level results in a system error with `exit`.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level and if the file address contained at the specified level is valid. If either condition is violated, control is transferred to the system error routine. In addition, the record ID on the specified level, if nonzero, is compared with the record ID in the record retrieved from file. The record code check on the specified level, if nonzero, is also compared to the record code check in the record. If either check fails, an error code is posted in the ECB level indicators.
- To ensure completion of the I/O, the requesting program must call `waitc` following `findc`. If an error has occurred, `waitc` will return a nonzero value.
- TPF transaction services processing affects `findc` processing in the following ways:
  - If a system error occurs because of one of the previous considerations, processing ends as if a rollback was issued.
  - The TPF system will first search for the record in the commit scope. If the record is not found, normal DASD retrieval will take place from virtual file access (VFA) or the DASD surface.

## Examples

The following example retrieves a data record from file on level D2 after making a call to FACS to calculate the file address.

```
#define VD1RI "#VD1RI "
#pragma map(get_file_addr,"FACS") /* map FACS to a function name */
#pragma linkage(get_file_addr,TPF,N) /* define appropriate linkage */
#include <tpfapi.h>
#include <tpfio.h>

/* set up storage */
struct TPF_regs *regs = (struct TPF_regs *) &(ecbptr()->ebx000);
:
regs->r6 = (long int) VD1RI /* record ID */
regs->r7 = (long int) &(ecbptr()->cel1fa2); /* lvl where file addr is */
regs->r0 = 10; /* ordinal number */
get_file_addr(regs); /* calculate fixed file address */
if (!regs->r0) /* FACS error? */
    if (regs->r7 == 1) /* invalid record ID */
        exit(0x12345);
    else
        {
            /* invalid ordinal number */
            serrc_op(SERRC_EXIT,0x12345,"INVALID ORDINAL NUMBER",NULL);
        }

findc(D2); /* find the record */
if (waitc()) /* abort on any errors */
    {
        serrc_op(SERRC_EXIT,0x12345,"I/O ERROR OCCURRED",NULL);
    }
/*
In TARGET(TPF) this used to be done with:
errno = 0x1234;
perror("I/O ERROR OCCURRED");
abort();
*/
}
```

## Related Information

- “findc\_ext—Find a Record with Extended Options” on page 176
- “find\_record—Find a Record” on page 178
- “finhc—Find and Hold a File Record” on page 185
- “finwc—Find a File Record and Wait” on page 189
- “fiwhc—Find and Hold a File Record and Wait” on page 193.

## findc\_ext—Find a Record with Extended Options

This function reads a record and attaches the block of working storage containing it to the specified data level of the ECB. The ECB must not be holding a storage block on the specified level.

This service finds a record that resides either in VFA or DASD.

### Format

```
#include <tpfio.h>
void findc_ext(enum t_lvl level, unsigned int ext);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The data record being retrieved is attached to this level.

#### ext

Sum of the following bit flags that are defined in `tpfio.h`.

#### FIND\_GDS

Use `FIND_GDS` to specify that the record to be read resides in a general file or general data set. If `FIND_GDS` is not specified, `findc_ext` accesses the record on the online database.

**Note:** If the flag is not needed, the default extended options flag, `FIND_DEFEXT`, should be coded. Consider using the `findc` function.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- The `findc_ext` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `findc_ext()` record X
  3. `tx_suspend_tpf()`
  4. `findc_ext()` record X.
- The specified CBRW is initialized to indicate that a block of storage is attached to the specified data level. Specifying an invalid data level results in a system error with `exit`.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level and if the file address contained at the specified level is valid. If either condition is violated, control is transferred to the system error routine. In addition, the record ID on the specified level, if nonzero, is compared with the record ID in the record retrieved from file. The record code check on the specified level, if nonzero, is also compared the record code check in the record. If either check fails, an error code is posted in the ECB level indicators.
- To ensure completion of the I/O, the requesting program must call `waitc` following `findc_ext`. If an error has occurred, `waitc` will return a nonzero value.

- TPF transaction services processing affects findc\_ext processing in the following ways:
  - If a system error occurs because of one of the previous considerations, processing ends as if a rollback was issued.
  - Finds from general files or general data sets are not considered part of the commit scope and are not affected by commit scope processing.
  - The TPF system will first search for the record in the commit scope. If the record is not found, normal DASD retrieval will take place from virtual file access (VFA) or the DASD surface.

## Examples

The following example retrieves a data record from a general data set to level D7. The file address has already been calculated and resides in the level D7 FARW. Note that FACS is not used to calculate file addresses for general files or general data sets.

```
#include <tpfio.h>
:
:
findc_ext(D7,FIND_GDS);
```

## Related Information

- “findc—Find a Record” on page 174
- “find\_record\_ext—Find a Record with Extended Options” on page 181
- “finhc\_ext—Find and Hold a File Record with Extended Options” on page 187
- “finwc\_ext—Find a File Record and Wait with Extended Options” on page 191
- “fiwhc\_ext—Find and Hold a File Record and Wait with Extended Options” on page 195.

## find\_record–Find a Record

This function retrieves a record from VFA or DASD. The CBRW specified by **level** must be unoccupied when `find_record` is called. If the operation is successful, the CBRW is occupied with a working storage block containing the record image at completion.

This function makes the equivalent of a `waitc` call.

For **HOLD**-type calls, the record address is placed in the record hold table and is available only to the issuing ECB.

## Format

```
#include <tpfio.h>
void *find_record(enum t_lvl level, const unsigned int *address,
                  const char *id, unsigned char rcc, enum t_act type);
```

### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter specifies an available FARW and CBRW for use by the system.

### address

A pointer to a file address from which the record will be retrieved.

**id** A pointer to a 2-character ID string that must match the ID characters in the record to be retrieved.

### rcc

An unsigned character that matches the record control check byte in the record to be retrieved.

### type

The record's intended hold status. This argument must belong to the enumeration type `t_act`, defined in `tpfio.h`.

### NOHOLD

The record address is not placed in the record hold table following I/O completion.

### HOLD

The record address is placed in the record hold table following I/O completion. The programmer is responsible for ensuring that the record is removed from the record hold table before calling the `exit` function.

## Normal Return

Pointer to the working storage block containing the record image following I/O completion.

## Error Return

Integer value of zero. Detailed error information can be found in the detail error byte (`ecbptr()->ce1sud[x]`), where `x` corresponds to the indicated data level.

## Programming Considerations

- The `find_record` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:



1. tx\_begin()
  2. file\_record() record X
  3. tx\_suspend\_tpf()
  4. find\_record() record X.
- The indicated CBRW must be unoccupied when the function is called. If not, control is transferred to the system error routine.
  - The indicated CBRW is occupied on return with a working storage block of the same size as the file record if the operation was successful.
  - All pending input operations are completed on return from this function. Pending output operations may not be completed.
  - For **HOLD**-type calls, the record address is placed in the record hold table and is available only to the issuing ECB. If the indicated record address is held by another ECB, the request is queued until the record is made available.
  - An error code is posted in the ECB level indicators if the values specified in arguments **id** and **rcc** do not match their corresponding fields in the retrieved record. Coding the **id** as **RECID\_RESET** and coding the **rcc** as \0 causes this check to be bypassed.
  - If the indicated record address is held by another ECB, this request is queued until the record is made available.
  - If a hardware error occurred, the system error routine issues a core dump and notifies the computer room agent set (CRAS) terminal.
  - If the FARW at the indicated level is already in a pre-initialized state (from a previous call to FACE or FACS, or general data set input), the programmer may want to bypass initializing the FARW values. If this is desired, code arguments **id** and **address** as NULL.
  - TPF transaction services processing affects find\_record processing in the following ways:
    - Additional checks need to be made to determine if the file address is held at the program level or at the commit scope level. Requests for file addresses held at the program level will be queued as usual. Requests for file addresses held outside of the commit scope will be queued. Requests for file addresses held in the commit scope will be serviced.
    - If a system error occurs because of one of the previous considerations, processing ends as if a rollback was issued.
    - The TPF system will first search for the record in the commit scope. If the record is not found, normal DASD retrieval will take place from virtual file access (VFA) or the DASD surface.

## Examples

The following example retrieves a forward chain IM record (rcc = 0) on D6 with hold.

```
#include <tpfio.h>
```

```
struct im0im
{
    char            im0bid[2];        /* record ID          */
    unsigned char    im0rcc;          /* record code check  */
    unsigned char    im0ctl;          /* control byte       */
    char            im0pgm[4];        /* program stamp      */
    unsigned long int im0fch;          /* forward chain      */
    unsigned long int im0bch;          /* backward chain     */
    short int        im0cct;          /* byte count         */
    .
    (data fields)
}
```

## find\_record

```
        .
    } *inm, *inm_chain;
    .
    .
    .
inm = ecbptr()->celcr5;          /* Base first in chain      */
if(!(inm_chain = find_record(D6, (const unsigned int *)&(inm->im0fch),
                                "IM",
                                '\0',
                                HOLD)))
{
    exit(0x12345);              /* Issue dump & exit if error */
}
else
{
    .
    .
    .
    .
    .
}
```

## Related Information

- “find\_record\_ext–Find a Record with Extended Options” on page 181
- “filec–File a Record: Basic” on page 152
- “unfrc–Unhold a File Record” on page 661.

## find\_record\_ext—Find a Record with Extended Options

This function retrieves a record from virtual file access (VFA) or DASD. The core block reference word (CBRW) specified by **level** or **decb** must be unoccupied when `find_record_ext` is called. If the operation is successful, the CBRW is occupied with a working storage block containing the record image at completion.

This function makes the equivalent of a `waitc` call unless otherwise requested.

For **HOLD**-type calls (including `HOLD_NOWAIT` and `HOLD_WAIT`), the record address is placed in the record hold table and is available only to the issuing ECB.

### Format

```
#include <tpfio.h>
void *find_record_ext(enum t_lvl level, const unsigned int *address,
                     const char *id, unsigned char rcc,
                     enum t_act type, unsigned int ext);
```

or

```
#include <tpfio.h>
void *find_record_ext(TPF_DECB *decb, TPF_FA8 *fa8,
                     const char *id, unsigned char rcc,
                     enum t_find_decb find_type, unsigned int ext);
```

#### level

One of 16 possible values representing a valid entry control block (ECB) data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter specifies an available file address reference word (FARW) and CBRW for use by the system.

#### decb

A pointer to a data event control block (DECB). This parameter specifies a FARW and CBRW for use by the system.

#### address

A pointer to a file address from which the record will be retrieved.

#### fa8

A pointer to an 8-byte file address from which the record will be retrieved.

**id** A pointer to a 2-character record ID string that must match the ID characters in the record to be retrieved. This check may be bypassed by coding the term **RECID\_RESET**, defined in `tpfio.h`.

#### rcc

An unsigned character that matches the record control check byte in the record to be retrieved.

#### type

The record's intended hold status. This parameter must belong to the enumeration type `t_act`, defined in `tpfio.h`.

#### NOHOLD

The record address is not placed in the record hold table following I/O completion.

#### HOLD

The record address is placed in the record hold table following I/O completion.

## find\_record\_ext

### find\_type

The hold and wait status of the record. The value of this parameter must belong to enumeration type `t_find_decb`, defined in `tpfio.h` and is one of the following:

#### NOHOLD\_NOWAIT

Do not wait for the I/O operation to be completed before returning to the calling program and the record address is not placed in the record hold table (RHT) following I/O completion. The status of the operation is unknown on return to the calling program. To ensure that the operation is completed, a `waitc` function must be called. After the `waitc` function, the CBRW at the specified DECB contains the storage address of the record.

#### HOLD\_NOWAIT

Do not wait for the I/O operation to be completed before returning to the calling program and the record address is placed in the RHT following I/O completion. However, the status of the operation is unknown on return to the calling program. To ensure that the operation is completed, a `waitc` function must be called. After the `waitc` function, the CBRW at the specified DECB contains the storage address of the record.

#### NOHOLD\_WAIT

The I/O operation is completed before returning to the calling program. The record will not be added to the RHT following I/O completion.

#### HOLD\_WAIT

The I/O operation is completed before returning to the calling program. The record will be added to the RHT following I/O completion.

### ext

Sum of the following bit flags, that are defined in `tpfio.h`.

### FIND\_GDS

Use `FIND_GDS` to specify that the record to be retrieved resides in a general file or general data set. If `FIND_GDS` is not specified, `find_record_ext` accesses the record on the online database.

**Note:** If the flag is not needed, the default extended options flag (`FIND_DEFEXT`) should be coded. Consider using the `find_record` function.

## Normal Return

A pointer to the working storage block containing the record image following I/O completion.

## Error Return

An integer value of zero. Detailed error information can be found in detail error byte `ecbptr()->celsud[x]`, where `x` corresponds to the indicated ECB data level, or the SUD field in the DECB.

## Programming Considerations

- The `find_record_ext` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `file_record()` record X
  3. `tx_suspend_tpf()`

4. find\_record\_ext() record X.

- The indicated CBRW must be unoccupied when the function is called. If not, control is transferred to the system error routine.
- The indicated CBRW is occupied on return with a working storage block of the same size as the file record if the operation was successful.
- All pending input operations are completed on return from this function. Pending output operations may not be completed.
- For **HOLD**-type calls, the record address is placed in the record hold table and is available only to the issuing ECB. If the indicated record address is held by another ECB, the request is queued until the record is made available.
- If a record hold was requested, the programmer is responsible for ensuring that the record is removed from the record hold table before calling the exit function.
- An error code is posted in the ECB data level or the DECB indicators if the values specified in parameters **id** and **rcc** do not match their corresponding fields in the retrieved record. Coding **id** as **RECID\_RESET** and coding **rcc** as \0 causes this check to be bypassed.
- If the indicated record address is held by another ECB, this request is queued until the record is made available.
- If a hardware error occurred, the system error routine issues a core dump and notifies the computer room agent set (CRAS) terminal.
- If the FARW at the indicated ECB data level or DECB is already in a pre-initialized state (from a previous call to FACE or FACS, or general data set input), you may want to bypass initializing the FARW values. If so, code parameters **id** and **address** as NULL.
- TPF transaction services processing affects find\_record\_ext processing in the following ways:
  - Additional checks need to be made to determine if the file address is held at the program level or at the commit scope level. Requests for file addresses held at the program level will be queued as usual. Requests for file addresses held outside of the commit scope will be queued. Requests for file addresses held in the commit scope will be serviced.
  - If a system error occurs because of one of the previous considerations, processing ends as if a rollback was issued.
  - Finds from general files or general data sets are not considered part of the commit scope and are not affected by commit scope processing.
  - The TPF system will first search for the record in the commit scope. If the record is not found, normal DASD retrieval will take place from virtual file access (VFA) or the DASD surface.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example retrieves a data record (**id** = CD, **rcc** = 0) from a general data set to level D7.

```
#include <tpfio.h>

unsigned file_ptr;
```

## find\_record\_ext

```

    :
ecbptr()->ebcfa7 = 0x0d0002f5; /* general file address to read */
find_record_ext(D7,(const unsigned int *)&ecbptr()->ebcfa7,
               CD,'\0',NOHOLD,FIND_GDS);

```

The following example retrieves and holds a data record with the file address and record ID specified in a DECB.

```

#include <tpfio.h>
TPF_DECB *decb
:
find_record_ext(decb, NULL, NULL, '\0', HOLD_WAIT, FIND_DEFEXT);

```

## Related Information

- “find\_record–Find a Record” on page 178
- “filec\_ext–File a Record with Extended Options: Basic” on page 154
- “unfrc\_ext–Unhold a File Record with Extended Options” on page 662.

See *TPF Application Programming* for more information about DECBs.

## finhc—Find and Hold a File Record

This function reads a record and attaches the block of working storage containing it to the specified data level of the entry control block (ECB). The ECB must not be holding a storage block on the specified level.

This service finds a record that resides either in VFA or on DASD.

The TPF system queues this request if the record is being held by another entry. All succeeding requests to hold this record are queued until the record is unheld.

### Format

```
#include <tpfio.h>
void finhc(enum t_lvl level);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The data record being retrieved is attached to this level.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The `finhc` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filec()` record X
  3. `tx_suspend_tpf()`
  4. `finhc()` record X.
- The specified CBRW is initialized to indicate that a block of storage is attached to the specified data level. Specifying an invalid data level results in a system error with `exit`.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level and if the file address contained at the specified level is valid. If either condition is violated, control is transferred to the system error routine. In addition, the record ID on the specified level, if nonzero, is compared with the record ID in the record retrieved from file. The record code check on the specified level, if nonzero, is also compared the record code check in the record. If either check fails, an error code is posted in the ECB level indicators.
- To ensure completion of the I/O, the requesting program must call `waitc` following `finhc`.
- TPF transaction services processing affects `finhc` processing in the following ways:
  - Additional checks need to be made to determine if the file address is held at the program level or at the commit scope level. Requests for file addresses held at the program level will be queued as usual. Requests for file addresses held outside of the commit scope will be queued. Requests for file addresses held in the commit scope will be serviced.

## finhc

- If a system error occurs because of one of the previous considerations, processing ends as if a rollback was issued.
- The TPF system will first search for the record in the commit scope. If the record is not found, normal DASD retrieval will take place from virtual file access (VFA) or the DASD surface.
- If a deadlock condition is detected on the ECB that issued this C function, a deadlock user exit is called with the ECB address and input/output block (IOB) address as an input. If the return code from the user exit is 0, processing continues as if the user exit was never called. If the return code from the user exit is 4, the ECB is scheduled to exit with dump D9. If the return code from the user exit is 8, CE1SUD and CE1SUG of the ECB will be set to CJCSUHRD and CJCSUDLK (that is, X'81') and the waiting IOB is removed from the system.

## Examples

The following example retrieves a data record from file on level D2 with hold. The file address has already been calculated and resides in the level D2 FARW.

```
#include <tpfio.h>
:
:
finhc(D2);
if (waitc())
{
    serrc_op(SERRC_EXIT,0x12345,"I/O ERROR OCCURRED",NULL);
/*
    In TARGET(TPF) this used to be done with:
    errno = 0x1234;
    perror("I/O ERROR OCCURRED");
    abort();
*/
}
```

## Related Information

- “finhc\_ext–Find and Hold a File Record with Extended Options” on page 187
- “find\_record–Find a Record” on page 178
- “findc–Find a Record” on page 174
- “finwc–Find a File Record and Wait” on page 189
- “fiwhc–Find and Hold a File Record and Wait” on page 193.



## finhc\_ext—Find and Hold a File Record with Extended Options

This function reads a record and attaches the block of working storage containing it to the specified data level of the entry control block (ECB). The ECB must not be holding a storage block on the specified level.

This service finds a record that resides either in VFA or on DASD.

TPF queues this request if the record is being held by another entry. All succeeding requests to hold this record are queued until the record is unheld.

### Format

```
#include <tpfio.h>
void finhc_ext(enum t_lvl level, unsigned int ext);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The data record being retrieved is attached to this level.

#### ext

Sum of the following bit flags that are defined in `tpfio.h`.

#### FIND\_GDS

Use `FIND_GDS` to specify that the record to be read resides in a general file or general data set. If `FIND_GDS` is not specified, `finhc_ext` accesses the record on the online database.

**Note:** If the flag is not needed, the default extended options flag (`FIND_DEFEXT`) should be coded. Consider using the `finhc` function.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- The `finhc_ext` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filec()` record X
  3. `tx_suspend_tpf()`
  4. `finhc_ext()` record X.
- The specified CBRW is initialized to indicate that a block of storage is attached to the specified data level. Specifying an invalid data level results in a system error with `exit`.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level and if the file address contained at the specified level is valid. If either condition is violated, control is transferred to the system error routine. In addition, the record ID on the specified level, if nonzero, is compared with the record ID in the record retrieved from file. The record code check on the

## finhc\_ext

specified level, if nonzero, is also compared the record code check in the record. If either check fails, an error code is posted in the ECB level indicators.

- To ensure completion of the I/O, the requesting program must call `waitc` following `finhc_ext`.
- TPF transaction services processing affects `finhc_ext` processing in the following ways:
  - Additional checks need to be made to determine if the file address is held at the program level or at the commit scope level. Requests for file addresses held at the program level will be queued as usual. Requests for file addresses held outside of the commit scope will be queued. Requests for file addresses held in the commit scope will be serviced.
  - If a system error occurs because of one of the previous considerations, processing ends as if a rollback was issued.
  - Finds from general files or general data sets are not considered part of the commit scope and are not affected by commit scope processing.
  - The TPF system will first search for the record in the commit scope. If the record is not found, normal DASD retrieval will take place from virtual file access (VFA) or the DASD surface.
  - If a deadlock condition is detected on the ECB that issued this C function, a deadlock user exit is called with the ECB address and input/output block (IOB) address as an input. If the return code from the user exit is 0, processing continues as if the user exit was never called. If the return code from the user exit is 4, the ECB is scheduled to exit with dump D9. If the return code from the user exit is 8, CE1SUD and CE1SUG of the ECB will be set to CJCSUHRD and CJCSUDLK (that is, X'81') and the waiting IOB is removed from the system.

## Examples

The following example retrieves a data record from a general data set to level D7. The file address has already been calculated and resides in the level D7 FARW. Note that FACS is not used to calculate file addresses for general files or general data sets.

```
#include <tpfio.h>
:
:
finhc_ext(D7,FIND_GDS);
if (waitc())
{
    serrc_op(SERRC_EXIT,0x12345,"I/O ERROR OCCURRED",NULL);
/*
    In TARGET(TPF) this used to be done with:
    errno = 0x1234;
    perror("I/O ERROR OCCURRED");
    abort();
*/
}
```

## Related Information

- “finhc—Find and Hold a File Record” on page 185
- “find\_record\_ext—Find a Record with Extended Options” on page 181
- “findc\_ext—Find a Record with Extended Options” on page 176
- “finwc\_ext—Find a File Record and Wait with Extended Options” on page 191
- “fiwhc\_ext—Find and Hold a File Record and Wait with Extended Options” on page 195.

## finwc—Find a File Record and Wait

This function reads a record and attaches the block of working storage containing it to the specified data level of the entry control block (ECB). The ECB must not be holding a storage block on the specified level.

This service finds a record that resides either in VFA or on an external device.

### Format

```
#include <tpfio.h>
void *finwc(enum t_lvl level);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The data record being retrieved is attached to this level.

### Normal Return

Pointer to the working storage block containing the retrieved record image.

### Error Return

NULL.

## Programming Considerations

- The `finwc` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filec()` record X
  3. `tx_suspend_tpf()`
  4. `finwc()` record X.
- The specified CBRW is initialized to indicate that a block of storage is attached to the specified data level. Specifying an invalid data level results in a system error with `exit`.
- TPF checks to determine if the ECB is holding a block of storage at the specified level, and if the file address contained at the specified level is valid. If either condition is violated, control is transferred to the system error routine. In addition, the record ID on the specified level, if nonzero, is compared with the record ID in the record retrieved from file. The record code check on the specified level, if nonzero, is also compared the record code check in the record. If either check fails, an error code is posted in the ECB level indicators.
- Calling this function resets the 500-millisecond program time-out.
- TPF transaction services processing affects `finwc` processing in the following ways:
  - If a system error occurs because of one of the previous considerations, processing ends as if a rollback was issued.
  - The TPF system will first search for the record in the commit scope. If the record is not found, normal DASD retrieval will take place from virtual file access (VFA) or the DASD surface.

## finwc

### Examples

The following example retrieves a data record from file on level D2. The file address has already been calculated and resides in the level D2 FARW. Control is returned to the operational program when the I/O is complete and the record has been attached to the specified level.

```
#include <tpfio.h>
struct im0im *inm;
:
inm = finwc(D2);
```

### Related Information

- “finwc\_ext–Find a File Record and Wait with Extended Options” on page 191
- “find\_record–Find a Record” on page 178
- “findc–Find a Record” on page 174
- “finhc–Find and Hold a File Record” on page 185
- “fiwhc–Find and Hold a File Record and Wait” on page 193.

## finwc\_ext–Find a File Record and Wait with Extended Options

This function reads a record and attaches the block of working storage containing it to the specified data level of the entry control block (ECB). The ECB must not be holding a storage block on the specified level.

This service finds a record that resides either in VFA or on an external device.

### Format

```
#include <tpfio.h>
void *finwc_ext(enum t_lvl level, unsigned int ext);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The data record being retrieved is attached to this level.

#### ext

Sum of the following bit flags that are defined in `tpfio.h`.

#### FIND\_GDS

Use `FIND_GDS` to specify that the record to be read resides in a general file or general data set. If `FIND_GDS` is not specified, `finwc_ext` accesses the record on the online database.

**Note:** If the flag is not needed, the default extended options flag (`FIND_DEFEXT`) should be coded. Consider using the `finwc` function.

### Normal Return

Pointer to the working storage block containing the retrieved record image.

### Error Return

NULL.

### Programming Considerations

- The `finwc_ext` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filec()` record X
  3. `tx_suspend_tpf()`
  4. `finwc_ext()` record X.
- The specified CBRW is initialized to indicate that a block of storage is attached to the specified data level. Specifying an invalid data level results in a system error with `exit`.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level and if the file address contained at the specified level is valid. If either condition is violated, control is transferred to the system error routine. In addition, the record ID on the specified level, if nonzero, is compared with the record ID in the record retrieved from file. The record code check on the specified level, if nonzero, is also compared with the record code check in the record. If either check fails, an error code is posted in the ECB level indicators.
- Calling this function resets the 500-millisecond program time-out.

## finwc\_ext

- TPF transaction services processing affects finwc\_ext processing in the following ways:
  - If a system error occurs because of one of the previous considerations, processing ends as if a rollback was issued.
  - Finds from general files or general data sets are not considered part of the commit scope and are not affected by commit scope processing.
  - The TPF system will first search for the record in the commit scope. If the record is not found, normal DASD retrieval will take place from virtual file access (VFA) or the DASD surface.

## Examples

The following example retrieves a data record from a general data set to level D7. The file address has already been calculated and resides in the level D7 FARW. Note that FACS is not used to calculate file addresses for general files or general data sets.

```
#include <tpfio.h>
:
:
inm = finwc_ext(D7,FIND_GDS);
```

## Related Information

- “finwc–Find a File Record and Wait” on page 189
- “find\_record\_ext–Find a Record with Extended Options” on page 181
- “findc\_ext–Find a Record with Extended Options” on page 176
- “finhc\_ext–Find and Hold a File Record with Extended Options” on page 187
- “fiwhc\_ext–Find and Hold a File Record and Wait with Extended Options” on page 195.

## fiwhc—Find and Hold a File Record and Wait

This function reads a record and attaches the block of working storage containing it to the specified data level of the ECB. The ECB must not be holding a storage block on the specified level.

This service finds a record that resides either in VFA or on an external device.

The control program queues this request if the record is being held by another entry. All succeeding requests to hold this record are queued until the record is unheld.

### Format

```
#include <tpfio.h>
void *fiwhc(enum t_lvl level);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The data record being retrieved is attached to this level.

### Normal Return

Pointer to the working storage block containing the retrieved record image.

### Error Return

NULL.

### Programming Considerations

- The `fiwhc` function cannot be issued on a record that is part of the suspended commit scope of an ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filec()` record X
  3. `tx_suspend_tpf()`
  4. `fiwhc()` record X.
- The specified CBRW is initialized to indicate that a block of storage is attached to the specified data level. Specifying an invalid data level results in a system error with `exit`.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level and if the file address contained at the specified level is valid. If either condition is violated, control is transferred to the system error routine. In addition, the record ID on the specified level, if nonzero, is compared with the record ID in the record retrieved from file. The record code check on the specified level, if nonzero, is also compared with the record code check in the record. If either check fails, an error code is posted in the ECB level indicators.
- Calling this service resets the 500-millisecond program time out.
- TPF transaction services processing affects `fiwhc` processing in the following ways:
  - Additional checks need to be made to determine if the file address is held at the program level or at the commit scope level. Requests for file addresses held at the program level will be queued as usual. Requests for file addresses held outside of the commit scope will be queued. Requests for file addresses held in the commit scope will be serviced.

## fiwhc

- If a system error occurs because of one of the previous considerations, processing ends as if a rollback was issued.
- The TPF system will first search for the record in the commit scope. If it is not found, normal DASD retrieval will take place from virtual file access (VFA) or the DASD surface.
- If a deadlock condition is detected on the ECB that issued this C function, a deadlock user exit is called with the ECB address and input/output block (IOB) address as an input. If the return code from the user exit is 0, processing continues as if the user exit was never called. If the return code from the user exit is 4, the ECB is scheduled to exit with dump D9. If the return code from the user exit is 8, CE1SUD and CE1SUG of the ECB will be set to CJCSUHRD and CJCSUDLK (that is, X'81') and the waiting IOB is removed from the system.

## Examples

The following example retrieves a data record from file on level D2 with hold. The file address has already been calculated and resides in the level D2 FARW. Control is returned to the operational program when the I/O is complete and the record has been attached to the specified level.

```
#include <tpfio.h>
struct im0im *inm;
:
inm = fiwhc(D2);
```

## Related Information

- “fiwhc\_ext–Find and Hold a File Record and Wait with Extended Options” on page 195
- “find\_record–Find a Record” on page 178
- “findc–Find a Record” on page 174
- “finhc–Find and Hold a File Record” on page 185
- “finwc–Find a File Record and Wait” on page 189.



## fiwhc\_ext–Find and Hold a File Record and Wait with Extended Options

This function reads a record and attaches the block of working storage containing it to the specified data level of the ECB. The ECB must not be holding a storage block on the specified level.

This service finds a record that resides either in VFA or on an external device.

The control program queues this request if the record is being held by another entry. All succeeding requests to hold this record are queued until the record is unheld.

### Format

```
#include <tpfio.h>
void *fiwhc_ext(enum t_lvl level, unsigned int ext);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The data record being retrieved is attached to this level.

#### ext

Logical or (v) of the following bit flags which are defined in `tpfio.h`.

#### FIND\_GDS

Use `FIND_GDS` to specify that the record to be read resides in a general file or general data set. If `FIND_GDS` is not specified, `fiwhc_ext` accesses the record on the online database.

**Note:** If the flag is not needed, the default extended options flag, (`FIND_DEFEXT`) should be coded. Consider using the `find_record` function.

### Normal Return

Pointer to the working storage block containing the retrieved record image.

### Error Return

NULL.

### Programming Considerations

- The `fiwhc_ext` function cannot be issued on a record that is part of the suspended commit scope of the ECB. The following sequence will cause a system error:
  1. `tx_begin()`
  2. `filec()` record X
  3. `tx_suspend_tpf()`
  4. `fiwhc_ext()` record X.
- The specified CBRW is initialized to indicate that a block of storage is attached to the specified data level. Specifying an invalid data level results in a system error with `exit`.
- The TPF system checks to determine if the ECB is holding a block of storage at the specified level and if the file address contained at the specified level is valid. If either condition is violated, control is transferred to the system error routine. In

## fiwhc\_ext

addition, the record ID on the specified level, if nonzero, is compared with the record ID in the record retrieved from file. The record code check on the specified level, if nonzero, is also compared with the record code check in the record. If either check fails, an error code is posted in the ECB level indicators.

- Calling this service resets the 500-millisecond program time out.
- TPF transaction services processing affects fiwhc\_ext processing in the following ways:
  - Additional checks need to be made to determine if the file address is held at the program level or at the commit scope level. Requests for file addresses held at the program level will be queued as usual. Requests for file addresses held outside of the commit scope will be queued. Requests for file addresses held in the commit scope will be serviced.
  - If a system error occurs because of one of the previous considerations, processing ends as if a rollback was issued.
  - Finds from general files or general data sets are not considered part of the commit scope and are not affected by commit scope processing.
  - The TPF system will first search for the record in the commit scope. If the record is not found, normal DASD retrieval will take place from virtual file access (VFA) or the DASD surface.
  - If a deadlock condition is detected on the ECB that issued this C function, a deadlock user exit is called with the ECB address and input/output block (IOB) address as an input. If the return code from the user exit is 0, processing continues as if the user exit was never called. If the return code from the user exit is 4, the ECB is scheduled to exit with dump D9. If the return code from the user exit is 8, CE1SUD and CE1SUG of the ECB will be set to CJCSUHRD and CJCSUDLK (that is, X'81') and the waiting IOB is removed from the system.

## Examples

The following example retrieves a data record from a general data set to level D7. The file address has already been calculated and resides in the level D7 FARW. Note that FACS is not used to calculate file addresses for general files or general data sets.

```
#include <tpfio.h>
:
:
inm = fiwhc_ext(D7,FIND_GDS);
```

## Related Information

- “fiwhc—Find and Hold a File Record and Wait” on page 193
- “find\_record\_ext—Find a Record with Extended Options” on page 181
- “findc\_ext—Find a Record with Extended Options” on page 176
- “finhc\_ext—Find and Hold a File Record with Extended Options” on page 187
- “finwc\_ext—Find a File Record and Wait with Extended Options” on page 191.

## flipc—Interchange the Status of Two Data Levels

This function interchanges or *flips* the contents of 2 block reference words (CBRWs), file address reference words (FARWs), file address reference word extensions, and detailed error bytes (SUDs, or CE1SUDs).

### Format

```
#include <tpfapi.h>
void      flipc(enum t_lvl level1, enum t_lvl level2);
```

#### level1

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This is the first data level that participates in the status interchange.

#### level2

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This is the second data level that participates in the status interchange.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- Specifying the same data level for both parameters results in no action being taken. Specifying an invalid data level results in a system error with exit.
- Neither level specified may have an I/O operation in progress where the status of the storage block is unknown until a subsequent `waitc` call.

## Examples

The following example interchanges CBRWs, FARWs, file address reference word extensions, and detail error bytes on levels D2 and DF.

```
#include <tpfapi.h>
:
flipc(D2,DF);
```

## Related Information

None.

## flushCache—Flush the Cache Contents

This function flushes the contents of the cache.

### Format

```
#include <c$cach.h>
long flushCache(const cacheToken *cache_to_flush);
```

#### cache\_to\_flush

The returned cache\_token from the newCache function that defined the logical record cache.

### Normal Return

#### CACHE\_SUCCESS

The function is completed successfully.

### Error Return

#### CACHE\_ERROR\_HANDLE

The cache\_token provided for the cache\_to\_flush parameter is not valid.

## Programming Considerations

The flush occurs immediately and without regard for any other applications that may also be using the cache.

## Examples

The following example flushes the contents of the file system INODE cache.

```
#include <c$cach.h>
#include <i$glue.h>

struct icontrl * contrl_ptr;      /* pointer file system control area */

/* get pointer to file system control area */

contrl_ptr = cinfo_fast_ss(CINFO_CMMZERO,
                          ecptr()->ceidbi );

/* if using INODE cache, call to flush INODE cache */

if (contrl_ptr->icontrl_icacheToken.token1 != 0)
    flushCache(&contrl_ptr->icontrl_icacheToken);
```

## Related Information

- “deleteCache—Delete a Logical Record Cache” on page 81
- “deleteCacheEntry—Delete a Cache Entry” on page 82
- “newCache—Create a New Logical Record Cache” on page 376
- “readCacheEntry—Read a Cache Entry” on page 413
- “updateCacheEntry—Add a New or Update an Existing Cache Entry” on page 673.

## fopen—Open a File

This function opens a file.

### Format

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

#### filename

The name of the file to be opened as a stream.

#### mode

The access mode for the stream.

This function opens the file specified by **filename** and associates a stream with it. Table 4 describes the values of the **mode** parameter.

Table 4. Values for the Mode Parameter

File Mode	Description
r	Open a text file for reading. (The file must exist.)
w	Open a text file for writing. If the file already exists, its contents are destroyed.
a	Open a text file in append mode for writing at the end of the file. The fopen function creates the file if it does not exist.
r+	Open a text file for both reading and writing. (The file must exist.)
w+	Open a text file for both reading and writing. If the file already exists, its contents are destroyed.
a+	Open a text file in append mode for reading or for updating at the end of the file. The fopen function creates the file if it does not exist.
rb	Open a binary file for reading. (The file must exist.)
wb	Open an empty binary file for writing. If the file already exists, its contents are destroyed.
ab	Open a binary file in append mode for writing at the end of the file. The fopen function creates the file if it does not exist.
r+b or rb+	Open a binary file for both reading and writing. (The file must exist.)
w+b or wb+	Open an empty binary file for both reading and writing. If the file already exists, its contents are destroyed.
a+b or ab+	Open a binary file in append mode for reading or for updating at the end of the file. The fopen function creates the file if it does not exist.

**Note:** The TPF file system does not use different file formats for binary and text files. Each line of a text file ends with an EBCDIC '\n' (0x15) character. Both text and binary files are treated as streams of bytes; there is no structure imposed on or conversion performed on the byte stream.

**Attention:** Use the w, w+, wb, w+b, and wb+ parameters with care; data in existing files of the same name will be lost.

*Text files* contain printable characters and control characters organized into lines. Each line ends with a new-line character.

*Binary files* contain a series of characters. Any additional interpretation or structure imposed on a binary file is the responsibility of the application.

## fopen

When you open a file with a, a+, ab, a+b, or ab+ mode, all write operations take place at the end of the file. Although you can reposition the file pointer using the fseek, fsetpos, or rewind function, the write functions move the file pointer back to the end of the file before they carry out any output operation. This action prevents you from overwriting existing data.

When you specify the update mode (using + in the second or third position), you can read from and write to the file. However, when switching between reading and writing, you must include an intervening positioning function such as fseek, fsetpos, rewind, or fflush. Output may immediately follow input if the end-of file (EOF) was detected.

## Normal Return

If successful, the fopen function returns a pointer to the object controlling the associated stream.

## Error Return

A NULL pointer return value indicates an error.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

When a new regular file is created, if the TPF\_REGFILE\_RECORD\_ID environment variable is set to a 2-character string, its value is used as the record ID for all pool file records that are allocated to store the contents of the file.

## Examples

The following example attempts to open a file for reading.

```
#include <stdio.h>

int main(void)
{
    FILE *stream;

    /* The following call opens a text file for reading */

    if ((stream = fopen("myfile.dat", "r")) == NULL)
        printf("Could not open data file for reading\n");
}
```

## Related Information

- “fclose—Close File Stream” on page 127
- “freopen—Redirect an Open File” on page 215.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## fprintf, printf, sprintf–Format and Write Data

This function formats and writes data to a stream.

### Format

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int sprintf(char *buffer, const char *format, ...);
```

#### stream

The stream to be written.

#### format

A specification of the output text format (see the information that follows).

#### buffer

The address of the first character in the array in which the output will be written.

.... Additional objects to be converted to output text.

#### ISO-C only

The fprintf() function is not available in the TARGET(TPF) C library. The printf() function in the TARGET(TPF) library is a limited and nonstandard substitute for the standard printf() function. The TARGET(TPF) library has no support for files, *stdout*, or file redirection.

The TARGET(TPF) version of the printf() function does not have the same effect as the ISO-C version. Results cannot be predicted if you use both versions in the same application.

The three related functions (fprintf, printf, and sprintf) are referred to as the *fprintf family*.

The fprintf function formats and writes output to **stream**. It converts each entry in the argument list, if any, and writes to the stream according to the corresponding format specification in **format**.

The printf function formats and writes output to the standard output stream, *stdout*.

#### TARGET(TPF) restrictions

The format controls for the TARGET(TPF) printf function are limited to those described in *SAA Common Programming Interface C Reference - Level 2*.

All TARGET(TPF) restrictions described for the puts function also apply to the printf function, which depends on the method of implementation at your installation.

**Note:** This function relies on the puts function to operate correctly. If printf is called and puts support does not exist, control is transferred to the system error routine.

## fprintf - printf - sprintf

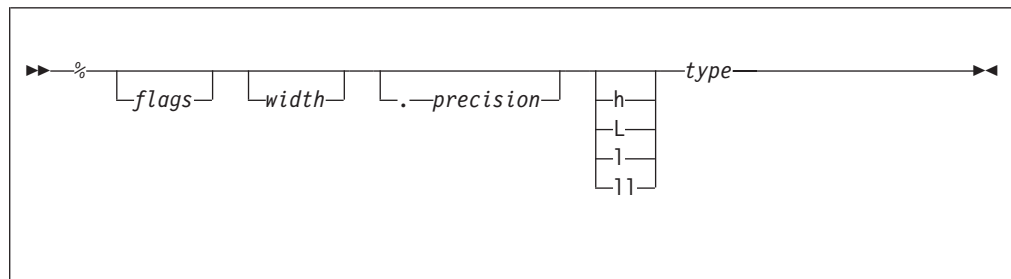
The `sprintf` function formats and stores a series of characters and values in the array pointed to by **buffer**. Any argument list is converted and put out according to the corresponding format specification in **format**. If the strings pointed to by **buffer** and **format** overlap, behavior is undefined.

The `fprintf` and `printf` functions have the same restriction as any write operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an end-of-file (EOF) has been reached.

The **format** consists of ordinary characters, escape sequences, and conversion specifications. The ordinary characters are copied in the order of their appearance. Conversion specifications, beginning with a percent sign (%), determine the output format for any *argument list* following **format**. The **format** can contain multibyte characters beginning and ending in the initial shift state.

When **format** includes the use of the optional prefix `ll` to indicate that the size expected is a long long data type, the corresponding value in the argument list should be a long long data type if correct output is expected.

The **format** is read from left to right. When the first format specification is found, the value of the first **argument** after **format** is converted and write according to the format specification. The second format specification causes the second argument after the **format** to be converted and write, and so on through the end of the **format**. If there are more arguments than there are format specifications, the extra arguments are evaluated and ignored. The results are undefined if there are not enough arguments for all the format specifications. The following shows the format specification.



Each field of 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`).

### The percent sign

If a percent sign (%) is followed by a character that has no meaning as a format field, the character is simply copied. For example, to print a percent sign character, use `%%`.

### The flag characters

The **flag** characters in Table 5 on page 203 are used for the justification of output and printing of signs, blanks, decimal points, octal and hexadecimal prefixes, and



the semantics for the `wchar_t` precision unit. Notice that more than one **flag** can appear in a format specification. This is an optional field.

*Table 5. Flag Characters for the fprintf Family*

Flag	Meaning	Default
–	Left-justify the result in 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 (–).
<i>blank</i> (' ')	Prefix the output value with a blank if the output value is signed and positive. The + flag overrides the <i>blank</i> flag if both appear, and a positive signed value will be written with a sign.	No blank.
#	When used with the o, x, or X formats, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.  When used with the f, e, or E formats, the # flag forces the output value to contain a decimal point in all cases.  The decimal point is sensitive to the LC_NUMERIC category of the same current locale.  When used with the g or G formats, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.	No prefix.  The decimal point appears only if digits follow it.  The decimal point appears only if digits follow it; trailing zeros are truncated.
	When used with the lS or S format, the # flag causes precision to be measured in wide characters.	Precision indicates the maximum number of bytes to be written.
0	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.

Do not use the # flag with c, lC, C, d, i, u, s, or p types.

### The width of the output

**Width** is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified **width**, blanks are added on the left or the right (depending on whether the – flag is specified) until the minimum width is reached.

**Width** never causes a value to be truncated; if the number of characters in the output value is greater than the specified **width**, or **width** is not given, all characters of the value are written (subject to the **precision** specification).

The **width** specification can be an asterisk (\*); if it is, an argument from the argument list supplies the value. The **width** argument must precede the value being formatted in the argument list. This is an optional field.

### The precision of the output

## fprintf - printf - sprintf

The **precision** specification is a nonnegative decimal integer preceded by a period. It specifies the number of characters to be written or the number of decimal places. Unlike the **width** specification, **precision** can cause the output value to be truncated or a floating-point value to be rounded.

The **precision** specification can be an asterisk (\*); if it is, an argument from the argument list supplies the value. The **precision** argument must precede the value being formatted in the argument list. The **precision** field is optional.

The interpretation of the **precision** value and the default when **precision** is omitted depend on the **type** as shown in Table 6.

Table 6. Precision Argument in the *fprintf* Family

Type	Meaning	Default
i d u o x X	<b>Precision</b> specifies the minimum number of digits to be written. If the number of digits in the argument is less than <b>precision</b> , the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds <b>precision</b> .	If <b>precision</b> is 0 or omitted entirely, or if the period (.) appears without a number following it, <b>precision</b> is set to 1.
f e E	<b>Precision</b> specifies the number of digits to be written after the decimal point. The last digit output is rounded.  The decimal point is sensitive to the LC_NUMERIC category of the current locale.	The default <b>precision</b> is 6. If <b>precision</b> is 0 or the period appears without a number following it, no decimal point is written.
g G	<b>Precision</b> specifies the maximum number of significant digits written.	All significant digits are written.
c	No effect.	The character is written.
C lc	No effect.	The wide character is written.
s	<b>Precision</b> specifies the maximum number of characters to be written. Characters in excess of <b>precision</b> are not written.	Characters are written until a null character is found.
S ls	<b>Precision</b> specifies the maximum number of bytes to be written. Bytes in excess of <b>precision</b> are not written; however, multibyte integrity is always preserved.	wchar_t characters are written until a null character is found.

### Optional prefix

The optional prefix is used to indicate the size of the argument expected:

- h** A prefix with integer types d, i, o, u, x, X, and n that specifies that the argument is 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.

The **l** prefix with the **c** type conversion specifier indicates that the argument is a wchar\_t. The **l** prefix with the **s** type conversion specifier indicates that the argument is a pointer to wchar\_t.

- L** A prefix with e, E, f, g, or G types that specifies that the argument is a long double value.

- 11 A prefix with integer types d, i, o, u, x, or XX that specifies that the integer is 64 bits long.

**Note:** If you pass a long double value and do not use the L qualifier, or if you pass a double value only and use the L qualifier, errors occur.

Table 7 shows the meaning of the type characters used in the precision argument.

*Table 7. Type Characters and Their Meanings*

Type	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	Double	Signed value having the form <code>[-]dddd.dddd</code> , where <code>dddd</code> 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 is equal to the requested precision.  The decimal point is sensitive to the <code>LC_NUMERIC</code> category of the current locale.
e	Double	Signed value having the form <code>[-]d.ddde[sign]ddd</code> , where <code>d</code> is a single-decimal digit, <code>dddd</code> is one or more decimal digits, <code>ddd</code> is two or more decimal digits, and <b>sign</b> is <code>+</code> or <code>-</code> .
E	Double	Identical to the <b>e</b> format except that E introduces the exponent, not e.
g	Double	Signed value output in f or e format. The e format is used only when the exponent of the value is less than <code>-4</code> or greater than <b>precision</b> . Trailing zeros are truncated and the decimal point appears only if one or more digits follow it.
G	Double	Identical to the g format, except that E introduces the exponent (where appropriate), not e.
D(n,p)	Decimal-type argument.	Fixed-point value consisting of an optional sign ( <code>+</code> or <code>-</code> ); a series of one or more decimal digits possibly containing a decimal point.
c	Character	Single character.
C or lc	Wide Character	The argument of <code>wchar_t</code> type is converted to an array of bytes representing a multibyte character as if by call to <code>wctomb()</code> .
s	String	Characters output up to the first null character ( <code>\0</code> ) or until <b>precision</b> is reached.
S or ls	Wide String	The argument is a pointer to an array of <code>wchar_t</code> type. Wide characters from the array are converted to multibyte characters up to and including a terminating null wide character. Conversion takes place as if by a call to <code>wcstombs()</code> with the conversion state described by the <code>mbstate_t</code> object initialized to 0. The result that is written out will not include the terminating null character.  If no precision is specified, the array contains a null wide character. If a precision is specified, it sets the maximum number of characters written, including shift sequences. A partial multibyte character cannot be written.
n	Pointer to integer	Number of characters successfully write so far to the <b>stream</b> or buffer; this value is stored in the integer whose address is given as the argument.

## fprintf - printf - sprintf

Table 7. Type Characters and Their Meanings (continued)

Type	Argument	Output Format
p	Pointer	Pointer to void converted to a sequence of printable characters. The output format is equivalent to type X.

## Normal Return

If successful the fprintf, printf, and sprintf functions return the number of characters written. The ending null character is not counted.

## Error Return

A negative value indicates that an output error has occurred.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

## Examples

The following example prints data using the printf function in a variety of formats.

```
#include <stdio.h>
int main(void)
{
    char ch = 'h', *string = "computer";
    int count = 234, hex = 0x10, oct = 010, dec = 10;
    double fp = 251.7366;
    unsigned int a = 12;
    float b = 123.45;
    int c;
    void *d = "a";

    printf("the unsigned int is %u\n\n",a);

    printf("the float number is %g, and %G\n\n",b,b);

    printf("RAY%n\n\n",&c);

    printf("last line prints %d characters\n\n",c);

    printf("Address of d is %p\n\n",d);

    printf("%d %d %06d %X %x %o\n\n",
        count, count, count, count, count, count);

    printf("1234567890123%n4567890123456789\n\n", &count);

    printf("Value of count should be 13; count = %d\n\n", count);

    printf("%10c%5c\n\n", ch, ch);

    printf("%25s\n%25.4s\n\n", string, string);

    printf("%f %.2f %e %E\n\n", fp, fp, fp, fp);

    printf("%i %i %i\n\n", hex, oct, dec);
}
```

## Output

```

the unsigned int is 12

the float number is 123.45 and 123.45

RAY

last line prints 3 characters

Address of d is DD72F9

234  +234  000234  EA  ea  352

12345678901234567890123456789

Value of count should be 13; count = 13

      h      h

              computer
              comp

251.736600  251.74  2.517366e+02  2.517366E+02

16  8  10

```

The following example shows the use of `printf` to print fixed-point decimal data types.

```

#include <stdio.h>
#include <decimal.h>

decimal(10,2) pd01 = -12.34d;
decimal(12,4) pd02 = 12345678.9876d;
decimal(31,10) pd03 = 123456789013579246801.9876543210d;

int main(void) {
    printf("pd01 %D(10,2)      = %D(10,2)\n", pd01);
    printf("pd02 %D( 12 , 4 ) = %D( 12 , 4 )\n", pd02);

    printf("pd01 %010.2D(10,2) = %010.2D(10,2)\n", pd01);
    printf("pd02 %20.2D(12,4)  = %20.2D(12,4)\n", pd02);
    printf("\n Give strange result if the specified size is wrong!\n");
    printf("pd03 %D(15,3)      = %D(15,3)\n\n", pd03);
}

```

### Output

```

pd01 %D(10,2)      = -12.34
pd02 %D( 12 , 4 ) = 12345678.9876
pd01 %010.2D(10,2) = -000012.34
pd02 %20.2D(12,4)  =          12345678.98

Give strange result if the specified size is wrong!
pd03 %D(15,3)      = -123456789013.579

```

The following example shows the use of the `sprintf` function to format and print various data.

```

#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "baltimore";
char c;

int main(void)

```

## fprintf - printf - sprintf

```
{
    c = 'l';
    i = 35;
    fp = 1.7320508;

    /* Format and print various data */
    j = sprintf(buffer, "%s\n", s);
    j += sprintf(buffer+j, "%c\n", c);
    j += sprintf(buffer+j, "%d\n", i);
    j += sprintf(buffer+j, "%f\n", fp);
    printf("string:\n%s\ncharacter count = %d\n", buff
}
```

### Output

```
string:
baltimore
l
35
1.732051
```

```
character count = 24
```

## Related Information

“fscanf, scanf, sscanf—Read and Format Data” on page 217.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## fputc—Write a Character

This function writes a character.

### Format

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

**c** The character to be written.

**stream**

The stream on which the character **c** is written.

This function converts **c** to an unsigned char, writes **c** to the output stream pointed to by **stream** at the current position, and advances the file position appropriately. The fputc function is identical to the putc function, but is always a function because it is not available as a macro.

If the stream is opened with one of the append modes, the character is appended to the end of the stream regardless of the current file position.

The fputc function has the same restriction as any write operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an end-of-file (EOF) has been reached.

### Normal Return

If successful, the fputc function returns the character written.

### Error Return

A return value of EOF indicates an error.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

## Examples

The following example writes the contents of the buffer to a file called myfile.dat. Because the output occurs as a side effect in the second expression of the for statement, the statement body is null.

```
#include <stdio.h>
#define NUM_ALPHA 26

int main(void)
{
    FILE * stream;
    int i;
    int ch;

    char buffer[NUM_ALPHA + 1] = "abcdefghijklmnopqrstuvwxyz";

    if (( stream = fopen("myfile.dat", "w"))!= NULL )
    {
        /* Put buffer into file */
        for ( i = 0; ( i < sizeof(buffer) ) &&
```

## fputc

```
        ((ch = fputc( buffer[i], stream)) != EOF ); ++i );  
    fclose( stream );  
}  
else  
    printf( "Error opening myfile.dat\n" );  
}
```

## Related Information

- “fgetc—Read a Character” on page 146
- “putc, putchar—Write a Character” on page 400.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



## fputs—Write a String

This function writes a string to a stream.

### Format

```
#include <stdio.h>
int fputs(const char *string, FILE *stream);
```

#### string

The address of the first character to be written.

#### stream

The stream on which the string is written.

This function writes the string pointed to by **string** to the output stream pointed to by **stream**. It does not write the terminating `\0` at the end of the string.

The `fputs` function has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an end-of-file (EOF) has been reached.

### Normal Return

If successful, the `fputs` function returns the number of bytes written.

### Error Return

If an error occurs, the `fputs` function returns EOF.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

## Examples

The following example writes a string to a stream.

```
#include <stdio.h>
#define NUM_ALPHA 26

int main(void)
{
    FILE * stream;
    int num;

    /* Do not forget that the '/' char occupies one character */
    static char buffer[NUM_ALPHA + 1] = "abcdefghijklmnopqrstuvwxyz";

    if ((stream = fopen("myfile.dat", "w")) != NULL )
    {
        /* Put buffer into file */
        if ( (num = fputs( buffer, stream )) != EOF )
        {
            /* Note that fputs() does not copy the /0 character */
            printf( "Total number of characters written to file = %i\n", num );
            fclose( stream );
        }
        else /* fputs failed */
            printf( "fputs failed" );
    }
}
```

## fputs

```
    }  
    else  
        printf( "Error opening myfile.dat" );  
}
```

## Related Information

- “fgets—Read a String from a Stream” on page 150
- “gets—Obtain Input String” on page 273
- “puts—Put String to Standard Output Stream” on page 402.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## fread—Read Items

This function reads items.

### Format

```
#include <stdio.h>
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

#### buffer

The address of the buffer into which the stream data will be read.

#### size

The number of bytes contained in each item to be read.

#### count

The number of items to be read.

#### stream

The stream to be read.

This function reads up to **count** items of **size** length from the input stream pointed to by **stream** and stores them in the given **buffer**. The file position indicator advances by the number of bytes read.

If there is an error during the read operation, the file position indicator is undefined. If a partial element is read, the value of the element is undefined.

The fread function has the same restriction as any read operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an end-of-file (EOF) has been reached.

### Normal Return

The fread function returns the number of complete items successfully read. If **size** or **count** is 0, fread returns 0 and the contents of the array and the state of the stream remain unchanged.

### Error Return

The ferror and feof functions are used to distinguish between a read error and an EOF. Note that EOF is only reached when an attempt is made to read beyond the last byte of data. Reading up to and including the last byte of data does **not** turn on the EOF indicator.

### Programming Considerations

None.

### Examples

The following example attempts to read NUM\_ALPHA characters from the myfile.dat file. If there are any errors with either the fread or fopen functions, a message is printed.

```
#include <stdio.h>
#define NUM_ALPHA 26

int main(void)
{
```

## fread

```
FILE * stream;
int num;          /* number of characters read from stream */

/* Do not forget that the '\0' char occupies one character too! */
char buffer[NUM_ALPHA + 1];
buffer[NUM_ALPHA+1] = '\0';

if (( stream = fopen("myfile.dat", "r"))!= NULL )
{
    num = fread( buffer, sizeof( char ), NUM_ALPHA, stream );
    if (num == NUM_ALPHA) { /* fread success */
        printf( "Number of characters read = %i\n", num );
        printf( "buffer = %s\n", buffer );
        fclose( stream );
    }
    else { /* fread() failed */
        if ( ferror(stream) ) /* possibility 1 */
            printf( "Error reading myfile.dat" );
        else if ( feof(stream)) { /* possibility 2 */
            printf( "EOF found\n" );
            printf( "Number of characters read %d\n", num );
            printf( "buffer = %.*s\n", num, buffer);
        }
    }
}
else
    printf( "Error opening myfile.dat" );
}
```

## Related Information

- “fopen—Open a File” on page 199
- “freopen—Redirect an Open File” on page 215
- “fwrite—Write Items” on page 239.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## freopen—Redirect an Open File

This function redirects an open file.

### Format

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

#### **filename**

The name of the file to replace an open stream.

#### **mode**

The access mode for the reopened stream.

#### **stream**

The stream to be reopened on the new file name.

This function closes the file currently associated with **stream** and pointed to by **stream**, opens the file specified by **filename**, and then associates the stream with it.

The `freopen` function opens the new file with the type of access requested by the **mode** argument. The **mode** argument is used as in the `fopen` function. See Table 4 on page 199 for a description of the **mode** parameter.

You can also use the `freopen` function to redirect the standard stream files `stdin`, `stdout`, and `stderr` to files that you specify. The file pointer parameter to the `freopen` function must point to a valid open file. If the file has been closed, the behavior is not defined.

You could use the following `freopen` call to redirect `stdout` to a file `A.B`:

```
freopen("A.B", "w", stdout);
```

### Normal Return

If successful, the `freopen` function returns the value of **stream**, the same value that was passed to it, and clears both the error and EOF indicators associated with the stream.

A failed attempt to close the original file is ignored.

### Error Return

If an error occurs when reopening the requested file, the `freopen` function closes the original file and returns a NULL pointer value.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

### Examples

The following example closes the **stream** data stream and reassigns its stream pointer:

```
#include <stdio.h>

int main(void)
```

## freopen

```
{
    FILE *stream, *stream2;

    stream = fopen("myfile.dat", "r");
    stream2 = freopen("myfile2.dat", "w+", stream);
}
```

**Note:** stream and stream2 will have the same value.

## Related Information

- “fclose—Close File Stream” on page 127
- “fopen—Open a File” on page 199.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## fscanf, scanf, sscanf—Read and Format Data

These functions read and format data.

### Format

```
#include <stdio.h>
int fscanf (FILE *:stream, const char *format, ...);
int scanf(const char *format, ...);
int sscanf(const char *buffer, const char *format, ...);
```

#### stream

The stream from which the text input will be read.

#### format

A specification of the format and conversions to be applied to the input text.

#### buffer

The string from which the input text will be read.

... Additional pointers to objects that will contain the converted data.

#### ISO-C only

The `fscanf` function is not available in the TARGET(TPF) C library. The `scanf()` function in the TARGET(TPF) library is a limited and nonstandard substitute for the standard `scanf()` function. The TARGET(TPF) library has no support for files, *stdin*, or file redirection.

The TARGET(TPF) version of the `scanf()` function does not have the same effect as the ISO-C version. Results cannot be predicted if you use both versions in the same application.

The `fscanf` function reads data from the current position of the specified **stream** into the locations given by the entries in the argument list, if any. The argument list, if it exists, follows the format string.

The `scanf` function reads data from standard input stream *stdin* into the locations given by each entry in the argument list. The argument list, if it exists, follows the format string.

#### TARGET(TPF) restrictions

The format controls for the TARGET(TPF) `scanf` function are limited to those described in *SAA Common Programming Interface C Reference - Level 2*.

All TARGET(TPF) restrictions described for the `gets` function also apply to the `scanf` function, which, therefore, depends on the method of implementation at your installation.

**Note:** This function relies on the `gets` function to operate correctly. If `scanf` is called and `gets` support does not exist, control is transferred to the system error routine.

The `sscanf` function reads data from **buffer** into the locations given by argument list. Reaching the end of the string pointed to by **buffer** is equivalent to the `fscanf`

## fscanf - scanf - sscanf

function reaching the end-of-file (EOF). If the strings pointed to by **buffer** and **format** overlap, the behavior is not defined.

The `fscanf` and `scanf` functions have the same restriction as any read operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an end-of-file (EOF) has been reached.

For all three functions, each entry in the argument list must be a pointer to a variable of a type that matches the corresponding conversion specification in **format**. If the types do not match, the results are not defined.

For all three functions, the **format** controls the interpretation of the argument list. The **format** can contain multibyte characters beginning and ending in the initial shift state.

The format string pointed to by **format** can contain one or more of the following:

- White-space characters, as specified by the `isspace` function, such as blanks and new-line characters. A white-space character causes the `fscanf`, `scanf`, and `sscanf` functions to read, but not store, all consecutive white-space characters in the input up to the next character that is not a white-space character. One white-space character in **format** matches any combination of white-space characters in the input.
- Characters that are not white space characters, except for the percent sign character (%). A non-white-space character causes the `fscanf`, `scanf`, and `sscanf` functions to read, but not store, a matching non-white-space character. If the next character in the input stream does not match, the function ends.
- Conversion specifications that are introduced by the percent sign (%). A conversion specification causes the `fscanf`, `scanf`, and `sscanf` functions to read and convert characters in the input into values of a conversion specifier. The value is assigned to an argument in the argument list.

All three functions read **format** from left to right. Characters outside of conversion specifications are expected to match the sequence of characters in the input stream; the matched characters in the input stream are scanned but not stored. If a character in the input stream conflicts with **format-string**, the function ends, ending with a matching failure. The conflicting character is left in the input stream as if it had not been read.

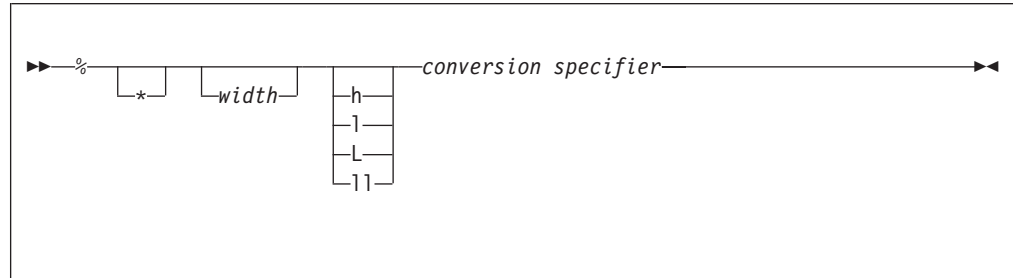
When the first conversion specification is found, the value of the first **input field** is converted according to the conversion specification and stored in the location specified by the first entry in the argument list. The second conversion specification converts the second input field and stores it in the second entry in the argument list, and so on through the end of **format-string**.

An **input field** is defined as:

- All characters until a white-space character (space, tab, or new-line) is found.
- All characters until a character is found that cannot be converted according to the conversion specification
- All characters until the **width** field is reached.



If there are too many arguments for the conversion specifications, the extra arguments are evaluated but otherwise ignored. The results are not defined if there are not enough arguments for the conversion specifications.



Each field of the conversion specification is a single character or a number signifying a particular format option. The **conversion specifier**, 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 conversion specification contains only the percent sign and a **conversion specifier** (for example, %s).

A detailed discussion of each field of the format specification follows.

To represent the character (%) in the format string, use a double percent sign (%%). A single percent sign % must always begin a conversion specifier. See Table 8 for a list of conversion specifiers.

An asterisk (\*) following the percent sign suppresses the assignment of the next input field, which is interpreted as a field of the specified **conversion specifier**. The field is scanned but not stored.

**width** is a positive decimal integer controlling the maximum number of characters to be read. No more than **width** characters are converted and stored at the corresponding **argument**.

Fewer than **width** characters are read if a white-space character (space, tab, or new-line) or a character that cannot be converted according to the given format occurs before **width** is reached.

Optional prefix l shows that you use the long version of the following **conversion specifier** while prefix h indicates that the short version will be used. The corresponding **argument** should point to a long or double object (for the l character), a long double object (for the L character), or a short object (with the h character). The l and h modifiers can be used with the d, i, o, x, and u **conversion specifiers**. The l modifier can also be used with the e, f, and g **conversion specifiers**. The L modifier can be used with the e, f, and g **conversion specifiers**. Note that the l modifier is also used with the c and s conversion specifiers to indicate a multibyte character or string. The l and h modifiers are ignored if specified for any other **conversion specifier**.

The **type** characters and their meanings are in Table 8.

Table 8. Conversion Specifiers in the fscanf and scanf functions

Conversion Specifier	Type of Input Expected	Type of Argument
d	Decimal integer.	Pointer to int.

## fscanf - scanf - sscanf

Table 8. Conversion Specifiers in the *fscanf* and *scanf* functions (continued)

Conversion Specifier	Type of Input Expected	Type of Argument
o	Octal integer.	Pointer to unsigned int.
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
D(n,p)	Fixed-point value consisting of an optional sign (+ or –); a series of one or more decimal digits possibly containing a decimal point.	Pointer to decimal.
c	(Can be used with the l modifier as lc). Character; white-space characters that are ordinarily skipped are read when c is specified.	Pointer to char large enough for input field.
C or lc	The input matches the number of multibyte characters specified by the field width. Each multibyte character in the sequence is converted to a wide character as if by a call to the mbstowcs function. The conversion state described by the mbstate_t object is initialized to zero before the first multibyte character is converted. The number of wide characters matched is specified by the field width (1 if no field width is present in the directive). The corresponding argument is a pointer to the initial element of an array of wchar_t large enough to accept the resulting sequence of wide characters. No null wide character is added.	C or lc uses a pointer to wchar_t string.
s	(Can be used with the l modifier as ls). String, which matches a sequence of multibyte characters that begins and ends in the initial shift state. None of the multibyte characters in the sequence are also single-byte white-space characters (as specified by the isspace function). Each multibyte character in the sequence is converted to a wide character as if by a call to the mbrtowc function, with the conversion state described by the mbstate_t object initialized to zero before the first multibyte character is converted.	Pointer to a character array large enough for input field, plus a terminating null character (\0) that is automatically appended.
S or lS	The corresponding argument is a pointer to the initial array of wchar_t large enough to accept the sequence and the terminating null wide character, which is added automatically.	S or lS uses a pointer to wchar_t string.

Table 8. Conversion Specifiers in the *fscanf* and *scanf* functions (continued)

Conversion Specifier	Type of Input Expected	Type of Argument
n	No input read from <b>stream</b> or <b>buffer</b> .	Pointer to <code>int</code> into which is stored the number of characters successfully read from the <b>stream</b> or <b>buffer</b> up to that point in the call.
p	Pointer to <code>void</code> converted to series of characters. The input format is equivalent to the <code>x</code> or <code>X</code> conversion specifier.	Pointer to <code>void</code> .
[ ]	<p>A nonempty sequence of bytes from a set of expected bytes (the <i>scanset</i>) that form the conversion specification. The conversion continues reading bytes until a failure to match or until an input failure.</p> <p>Consider the following conditions:</p> <p>[<sup>^</sup>bytes]. In this case, the <i>scanset</i> contains all bytes that do not appear between the circumflex and the right square bracket.</p> <p>[abc] or [<sup>^</sup>abc.] In both these cases the right square bracket is included in the <i>scanset</i> (in the first case: ]abc and in the second case, <i>not</i> ]abc).</p> <p>[a–z] The – is in the <i>scanset</i>; the characters b through y are <b>not</b> in the <i>scanset</i>.</p> <p>The code point for the square brackets ([ and ]) and the caret (^) vary among the EBCDIC encoded character sets. The default C locale expects these characters to use the code points for encoded character set Latin-1 / Open Systems 1047. Conversion proceeds 1 byte at a time; there is no conversion to wide characters.</p>	Pointer to the initial byte of an array of <code>char</code> , signed <code>char</code> , or unsigned <code>char</code> , large enough to accept the sequence and a terminating byte, that will be added automatically.

To read strings not delimited by space characters, substitute a set of characters in square brackets ([ ]) for the **s** (string) conversion specifier. The corresponding input field is read up to the first character that does **not** appear in the bracketed character set. If the first character in the set is a logical not (^), the effect is reversed: the input field is read up to the first character that **does** appear in the rest of the character set.

To store a string without storing an ending null character (\0), use the specification `%ac`, where *a* is a decimal integer. In this instance, the `c` conversion specifier means that the argument is a pointer to a character array. The next *a* characters are read from the input stream into the specified location and no null character is added.

The input for a `%x` conversion specifier is interpreted as a hexadecimal number.

All three functions (`fscanf`, `scanf`, and `sscanf`) scan each input field character by character. The function might stop reading a particular input field either before it reaches a space character, when the specified **width** is reached, or when the next

## fscanf - scanf - sscanf

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 character that is not read. The conflicting character, if there is one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on the input stream.

## Normal Return

All three functions (fscanf, scanf, and sscanf) return the number of input items that were successfully matched and assigned. The return value does not include conversions that were performed but not assigned (for example, suppressed assignments).

## Error Return

The functions return EOF if there is an input failure before any conversion or if EOF is reached before any conversion. Therefore, a return value of 0 means that no fields were assigned; there was a matching failure before any conversion. Also, if there is an input failure, the file error indicator is set, which is not the case for a matching failure.

The ferror and feof functions are used to distinguish between a read error and an EOF.

**Note:** EOF is reached only when an attempt is made to read beyond the last byte of data.

Reading up to and including the last byte of data does **not** turn on the EOF indicator.

## Programming Considerations

None.

## Examples

The following example scans various types of data.

```
#include <stdio.h>

int main(void)
{
    int i;
    float fp;
    char c, s[81];

    printf("Enter an integer, a real number, a character "
           "and a string : \n");
    if (scanf("%d %f %c %s", &i, &fp, &c, s) != 4)
        printf("Not all of the fields were assigned\n");
    else
    {
        printf("integer = %d\n", i);
        printf("real number = %f\n", fp);
        printf("character = %c\n", c);
        printf("string = %s\n", s);
    }
}
```

### Output

If input is: 12 2.5 a yes, then output would be:

```

Enter an integer, a real number, a character and a string:
integer = 12
real number = 2.500000
character = a
string = yes

```

The following example converts a hexadecimal integer to a decimal integer. The while loop ends if the input value is not a hexadecimal integer.

```

#include <stdio.h>

int main(void)
{
    int number;

    printf("Enter a hexadecimal number or anything else to quit:\n")
    while (scanf("%x",&number))
    {
        printf("Hexadecimal Number = %x\n",number);
        printf("Decimal Number      = %d\n",number);
    }
}

```

### Output

If input is: 0x231 0xf5e 0x1 q, output would be:

```

Enter a hexadecimal number or anything else to quit:
Hexadecimal Number = 231
Decimal Number      = 561
Hexadecimal Number = f5e
Decimal Number      = 3934
Hexadecimal Number = 1
Decimal Number      = 1

```

The following example shows the use of scanf to input fixed-point decimal data types.

```

#include <stdio.h>
#include <decimal.h>

decimal(15,4) pd01;
decimal(10,2) pd02;
decimal(5,5) pd03;

int main(void) {
    printf("\nFirst time :-----\n");
    printf("Enter three fixed-point decimal number\n");
    printf("  (15,4) (10,2) (5,5)\n");
    if (scanf("%D(15,4) %D(10,2) %D(5,5)", &pd01, &pd02, &pd03) != 3) {
        printf("Error found in scanf\n");
    } else {
        printf("pd01 = %D(15,4)\n", pd01);
        printf("pd02 = %D(10,2)\n", pd02);
        printf("pd03 = %D(5,5)\n", pd03);
    }
    printf("\nSecond time :-----\n");
    printf("Enter three fixed-point decimal number\n");
    printf("  (15,4) (10,2) (5,5)\n");
    if (scanf("%D(15,4) %D(10,2) %D(5,5)", &pd01, &pd02, &pd03) != 3) {
        printf("Error found in scanf\n");
    } else {
        printf("pd01 = %D(15,4)\n", pd01);
        printf("pd02 = %D(10,2)\n", pd02);
    }
}

```

## fscanf - scanf - sscanf

```
        printf("pd03 = %D(5,5)\n", pd03);
    }
    return(0);
}
```

### Output

```
First time :-----
Enter three fixed-point decimal number
(15,4) (10,2) (5,5)
12345678901.2345 -987.6 .24680
pd01 = 12345678901.2345
pd02 = -987.60
pd03 = 0.24680

Second time :-----
Enter three fixed-point decimal number
(15,4) (10,2) (5,5)
123456789013579.24680 123.4567890 987
pd01 = 12345678901.3579
pd02 = 123.45
pd03 = 0.98700
```

The next example opens the file myfile.dat for reading and then scans this file for a string, a long integer value, a character, and a floating-point value.

```
#include <stdio.h>
#define MAX_LEN 80

int main(void)
{
    FILE *stream;
    long l;
    float fp;
    char s[MAX_LEN + 1];
    char c;

    stream = fopen("myfile.dat", "r");

    /* Put in various data. */
    fscanf(stream, "%s", &s[0]);
    fscanf(stream, "%ld", &l);
    fscanf(stream, "%c", &c);
    fscanf(stream, "%f", &fp);

    printf("string = %s\n", s);
    printf("long double = %ld\n", l);
    printf("char = %c\n", c);
    printf("float = %f\n", fp);
}
```

### Output

If myfile.dat contains abcdefghijklmnopqrstuvwxyz 343.2, the expected output is:

```
string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
```

The following example uses the sscanf function to read various data from string tokenstring and then displays the data.

```
#include <stdio.h>
#define SIZE 81

int main(void)
```

```

{
char *tokenstring = "15 12 14";
int i;
float fp;
char s[SIZE];
char c;

    /* Input various data */
    printf("No. of conversions=%d\n",
        sscanf(tokenstring, "%s %c%d%f", s, &c, &i, &fp));

    /* If there were no space between %s and %c,
    /* sscanf would read the first character following
    /* the string, which is a blank space.
    /*

    /* Display the data */
    printf("string = %s\n",s);
    printf("character = %c\n",c);
    printf("integer = %d\n",i);
    printf("floating-point number = %f\n",fp);
}

```

### Output

```

No. of conversions = 4
string = 15
character = 1
integer = 2
floating-point number = 14.000000

```

## Related Information

"fprintf, printf, sprintf—Format and Write Data" on page 201.

See Appendix E, "Programming Support for the TPF File System" on page 1405 for more information about TPF File System C Functions.

## fseek—Change File Position

This function sets a file position.

### Format

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int origin);
```

#### stream

The stream for which the current position will be set.

#### offset

The distance in bytes from the origin.

#### origin

A position in the stream.

The **origin** must be one of the following constants defined in the `stdio.h` header file:

Origin	Definition
<b>SEEK_SET</b>	Beginning of the file.
<b>SEEK_CUR</b>	Current position of the file pointer.
<b>SEEK_END</b>	End of file.

This function changes the current file position associated with **stream** to a new location in the file. The next operation on the stream takes place at the new location. On a stream open for update, the next operation can be either a reading or a writing operation.

**Note:** If you specify `SEEK_CUR`, any characters that were pushed back by the `ungetc` or `ungetwc` functions will have backed up the current position of the file pointer, which is the starting point of the seek. The seek will discard any pushed-back characters before repositioning, but the starting point will still be affected. For more information about calling the `fseek` after an `ungetc` function see “`ungetc`—Push Character to Input Stream” on page 664. You can calculate your own offsets. If the offset exceeds the end-of-file (EOF) and you subsequently write new data, your file is extended with nulls.

Attempting to reposition before the start of the file causes the `fseek` function to fail.

**Attention:** Repositioning in a wide-oriented file and performing updates is not recommended because it is not possible to predict if your update will overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The following data expects the shift-out to be there, so it is not valid if it is treated as if in the initial shift state. Repositioning to the end of the file and adding new data is safe.

If successful, the `fseek` function clears the EOF indicator even when **origin** is `SEEK_END` and cancels the effect of any preceding `ungetc` or `ungetwc` function on the same stream.

If the call to the `fseek` function or the `fsetpos` function is not valid, the call is treated as a flush and the `ungetc` characters are discarded.



## Normal Return

If it successfully moves the pointer, the fseek function returns a zero value.

## Error Return

A nonzero return value indicates an error. On devices that cannot seek, such as terminals and printers, the return value is nonzero.

## Programming Considerations

None.

## Examples

The following example opens a myfile.dat file for reading. After performing input operations (not shown), the file pointer is moved to the beginning of the file.

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    int result;

    if (stream = fopen("myfile.dat", "r"))
    { /* successful */

        if (fseek(stream, 0L, SEEK_SET)); /* moves pointer to */
                                           /* the beginning of the file */
        { /* if not equal to 0
            then error ... */
        }
        else {
            /* fseek() successful */
        }
    }
}
```

## Related Information

- “ftell—Get Current File Position” on page 234
- “ungetc—Push Character to Input Stream” on page 664.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## fsetpos—Set File Position

This function sets a file position.

### Format

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

#### **stream**

The stream for which the current file position will be set.

#### **pos**

The new file position for the stream.

This function moves the file position associated with **stream** to a new location in the file according to the value of the object pointed to by **pos**. The value of **pos** must be obtained by a call to the `fgetpos` library function. If successful, the `fsetpos` function clears the end-of file (EOF) indicator and cancels the effect of any previous `ungetc` function on the same stream.

If the call to the `fsetpos` function is not valid, the call is treated as a flush and the `ungetc` characters are discarded.

The `fsetpos` function handles double-byte character set (DBCS) state information for wide-oriented files. The DBCS shift state is set to the state saved by the `fsetpos` function. If the record has been updated in the meantime, the shift state may be incorrect.

After the `fsetpos` call, the next operation on a stream in update mode may be input or output.

**Attention:** Repositioning in a wide-oriented file and performing updates is not recommended because it is not possible to predict if your update will overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The following data expects the shift-out to be there, so it is not valid if it is treated as if in the initial shift state. Repositioning to the end of the file and adding new data is safe.

### Normal Return

If the `fsetpos` function successfully changes the current position of the file, it returns a zero value.

### Error Return

If there is an error, `errno` is set and a nonzero value is returned.

### Programming Considerations

None.

### Examples

The following example opens a file called `myfile.dat` for reading. After performing input operations (not shown), the file pointer is moved to the beginning of the file and reads the first byte again.

```
#include <stdio.h>

int main(void)
```

```

{
    FILE *stream;
    int retcode;
    fpos_t pos, pos1, pos2, pos3;
    char ptr[20]; /* existing file 'myfile.dat' has 20 byte records */

    /* Open file, get position of file pointer, and read first record */

    stream = fopen("myfile.dat", "rb");
    fgetpos(stream,&pos);
    pos1 = pos;
    if (!fread(ptr,sizeof(ptr),1,stream))
        printf("fread error\n");

    /* Perform a number of read operations. The value of 'pos'
       changes if 'pos' is passed to fgetpos() */
    :
    /* Re-set pointer to start of file and re-read first record */

    fsetpos(stream,&pos1);
    if (!fread(ptr,sizeof(ptr),1,stream))
        printf("fread error\n");

    fclose(stream);
}

```

## Related Information

- “fgetpos—Get File Position” on page 148
- “ftell—Get Current File Position” on page 234
- “rewind—Set File Position to Beginning of File” on page 431
- “ungetc—Push Character to Input Stream” on page 664.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## fstat—Get Status Information about a File

This function gets status information about a file.

### Format

```
#include <sys/stat.h>
int fstat(int fildes, struct stat *info);
```

#### **fildes**

An open file descriptor.

#### **info**

The address of an object of type `struct stat` in which file status will be returned.

This function gets status information about the file specified by the open file descriptor, **fildes**, and stores it in the area of memory indicated by the **info** argument. The status information is returned in a `stat` structure as defined in the `sys/stat.h` header file.

### Normal Return

If successful, the `fstat` function returns a 0 value.

### Error Return

If unsuccessful, the `fstat` function returns `-1` and sets `errno` to one of the following:

<b>EBADF</b>	<b>fildes</b> is not a valid open file descriptor.
<b>EINVAL</b>	<b>info</b> is a null pointer.

## Programming Considerations

See Table 15 on page 513 for more information about the `stat` data structure, which is defined in the `sys/stat.h` header file.

## Examples

The following example gets status information for the file called `temp.file`.

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

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

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        if (fstat(fd, &info) != 0)
            perror("fstat() error");
        else {
            puts("fstat() returned:");
            printf("    inode:  %d\n",      (int) info.st_ino);
            printf("    dev id:  %.8x\n", info.st_dev);
            printf("    mode:   %.8x\n", info.st_mode);
        }
    }
}
```

```

        printf("    links:  %d\n",      info.st_nlink);
        printf("        uid:  %d\n",    (int) info.st_uid);
        printf("        gid:  %d\n",    (int) info.st_gid);
        printf("modified:  %s",        ctime(&info.st_mtime));
    }
    close(fd);
    unlink(fn);
}
}

```

### Output

```

fstat returned:
inode:    3057
dev id:   C2E2E240
mode:     02000080
links:    1
uid:      256
gid:      257
modified:  Fri Jan 16 16:03:16 1998

```

## Related Information

- “fcntl—Control Open File Descriptors” on page 129
- “lstat—Get Status of a File or Symbolic Link” on page 317
- “open—Open a File” on page 380.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## fsync—Write Changes to Direct Access Storage

This function writes changes to direct-access storage.

### Format

```
#include <unistd.h>
int fsync(int fildes);
```

#### **fildes**

An open file descriptor for the file to be synchronized.

This function transfers all data for the file indicated by the open file descriptor, **fildes**, to the storage device associated with **fildes**. The `fsync` function does not return until the transfer has been completed or an error is detected.

### Normal Return

If successful, the `fsync` function returns a 0 value.

### Error Return

If unsuccessful, the `fsync` function returns `-1` and sets `errno` to one of the following:

**EBADF**            **fildes** is not a valid open file descriptor.  
**EINVAL**          The file cannot be synchronized.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

## Examples

The following example shows one use of the `fsync` function.

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

#define mega_string_len 250000

main() {
    char *mega_string;
    int fd, ret;
    char fn[]="FSYNC.FILE";

    if ((mega_string = malloc(mega_string_len)) == NULL)
        perror("malloc() error");
    else if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        memset(mega_string, 's', mega_string_len);
        if ((ret = write(fd, mega_string, mega_string_len)) == -1)
            perror("write() error");
        else {
            printf("write() wrote %d bytes\n", ret);
            if (fsync(fd) != 0)
                perror("fsync() error");
            else if ((ret = write(fd, mega_string, mega_string_len)) == -1)
```

```
        perror("write() error");
    else
        printf("write() wrote %d bytes\n", ret);
    }
    close(fd);
    unlink(fn);
}
```

**Output**

```
write() wrote 250000 bytes
write() wrote 250000 bytes
```

**Related Information**

- “open—Open a File” on page 380
- “write—Write Data to a File Descriptor” on page 696.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## ftell—Get Current File Position

This function gets a current file position.

### Format

```
#include <stdio.h>
long int ftell(FILE *stream);
```

#### **stream**

The stream whose current file position will be returned.

This function obtains the current value of the file position indicator for the stream pointed to by **stream**.

### Normal Return

If successful, the `ftell` function returns the relative byte offset from the beginning of the file.

### Error Return

If unsuccessful, the `ftell` function returns `-1` and sets `errno` to a positive value.

## Programming Considerations

None.

## Examples

The following example opens a `myfile.dat` file for reading. The current file pointer position is stored in variable **pos**.

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    long int pos;

    stream = fopen("myfile.dat", "rb");

    /* The value returned by ftell can be used by fseek()
       to set the file pointer if 'pos' is not -1          */

    if ((pos = ftell(stream)) != EOF)
        printf("Current position of file pointer found\n");
    fclose(stream);
}
```

## Related Information

- “`fgetpos`—Get File Position” on page 148
- “`fopen`—Open a File” on page 199
- “`fseek`—Change File Position” on page 226
- “`fsetpos`—Set File Position” on page 228.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



## ftok—Generate a Token

This function returns a key (token) based on the specified path name and integer. The key can be used in subsequent calls to the `shmget` function.

### Format

```
#include <sys/ipc.h>
key_t    ftok(const char *path,
               int id);
```

#### **path**

The path name of an existing file.

**id** A nonzero integer that is used to generate the key that is returned by this function. Only the low-order 8 bits are used to generate the key.

### Normal Return

If successful, the `ftok` function returns a key that is specified as the **key** parameter in subsequent calls to the `shmget` function.

### Error Return

If unsuccessful, the `ftok` function returns a value of `-1` and sets `errno` to one of the following:

**EACCES** Search permission is denied for a path name component.

**EINVAL** The low-order 8 bits of the **id** parameter are zero.

**ELOOP** Too many symbolic links were found when resolving the **path** parameter.

#### **ENAMETOOLONG**

The length of the **path** parameter exceeds `PATH_MAX`, a path name component is longer than `NAME_MAX`, or when resolving the path name of a symbolic link, the length of an intermediate result exceeds `PATH_MAX`. `PATH_MAX` and `NAME_MAX` are defined in the `limits.h` header file.

**ENOENT** A path name component does not name an existing file or the **path** parameter is an empty string.

**ENOTDIR** A path name component is not a directory.

## Programming Considerations

- The `ftok` function returns the same key value for all paths that name the same file when called with the same value specified for the **id** parameter. If a different value is specified for the **id** parameter or a different file is specified, a different key value is returned.
- Only the low-order 8 bits of the **id** parameter are significant.

## Examples

The following example generates a key.

```
#include <sys/ipc.h>
int main()
{
```

## ftok

```
key_t key;  
  
key = ftok("/usr/testfile",3);  
}
```

## Related Information

- “shmat—Attach Shared Memory” on page 476
- “shmctl—Shared Memory Control” on page 478
- “shmdt—Detach Shared Memory” on page 481
- “shmget—Allocate Shared Memory” on page 483.

## ftruncate—Truncate a File

This function truncates a file.

### Format

```
#include <unistd.h>
int ftruncate(int fildes, off_t length);
```

#### **fildes**

An open file descriptor for the file whose size will be changed.

#### **length**

The new size of the file.

This function truncates the file indicated by the open file descriptor, **fildes**, to the indicated **length**. **fildes** must be a regular file or character special file that is open for writing. If the file size exceeds **length**, any extra data is discarded. If the file size is smaller than **length**, bytes between the old and new lengths are read as zeros. A change to the size of the file has no impact on the file offset.

#### TPF deviation from POSIX

The TPF system updates the `st_mtime` field only when the file is closed. The TPF system does not support the `ctime` time stamp.

If the `ftruncate` function is not successful, the file is not changed.

### Normal Return

If successful, the `ftruncate` function returns a 0 value.

### Error Return

If unsuccessful, the `ftruncate` function returns `-1` and sets `errno` to one of the following:

<b>EBADF</b>	<b>fildes</b> is not a valid open file descriptor.
<b>EINVAL</b>	<b>fildes</b> does not refer to a regular file or character special file, it is opened as read-only, or the length specified is incorrect.

### Programming Considerations

- The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.
- You may or may not be able to truncate a character special file depending on the type of device and the device driver. If the file cannot be truncated, the `ftruncate` function fails with `errno` set to `EINVAL`.

### Examples

The following example shows one use of the `ftruncate` function.

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

## ftruncate

```
#define string_len 1000

main() {
    char *mega_string;
    int fd, ret;
    char fn[]="write.file";
    struct stat st;

    if ((mega_string = malloc(string_len)) == NULL)
        perror("malloc() error");
    else if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        memset(mega_string, '0', string_len);
        if ((ret = write(fd, mega_string, string_len)) == -1)
            perror("write() error");
        else {
            printf("write() wrote %d bytes\n", ret);
            fstat(fd, &st);
            printf("the file has %ld bytes\n", (long) st.st_size);
            if (ftruncate(fd, 1) != 0)
                perror("ftruncate() error");
            else {
                fstat(fd, &st);
                printf("the file has %ld bytes\n", (long) st.st_size);
            }
        }
        close(fd);
        unlink(fn);
    }
}
```

### Output

```
write() wrote 1000 bytes
the file has 1000 bytes
the file has 1 bytes
```

## Related Information

“open—Open a File” on page 380.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## fwrite—Write Items

This function writes items.

### Format

```
#include <stdio.h>
size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);
```

#### buffer

The address of the data to be written.

#### size

The size of each item to be written.

#### count

The number of items to be written.

#### stream

The stream to be written.

This function writes up to **count** items of size **size** from the location pointed to by **buffer** to the stream pointed to by **stream**.

Because the fwrite function may buffer output before writing it out to the stream, data from previous fwrite calls may be lost whereby a subsequent call to the fwrite function causes a failure when the buffer is written to the stream.

The fwrite function has the same restriction as any write operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an end-of-file (EOF) has been reached.

### Normal Return

If successful, the fwrite function returns the number of items that were written. If there is no error, the return value is the same as **count**.

### Error Return

The return value can be smaller than **count** only if a write error occurs.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

### Examples

The following example writes NUM long integers to a stream in binary format. It checks that the fopen function is successful and that 100 items are written to the stream.

```
#include <stdio.h>
#define NUM 100

int main(void)
{
    FILE *stream;
    long list[NUM];
    int numwritten, number;
```

## **fwrite**

```
if((stream = fopen("myfile.dat", "w+b")) != NULL )
{
    for (number = 0; number < NUM; ++number)
        list[number] = number;
    numwritten = fwrite(list, sizeof(long), NUM, stream);
    printf("number of long characters written is %d\n",numwritten);
}
else
    printf("fopen error\n");
}
```

## **Related Information**

- “fopen—Open a File” on page 199
- “freopen—Redirect an Open File” on page 215
- “fread—Read Items” on page 213.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## gdsnc–Get Data Set Entry

This function initializes an entry control block (ECB) data level or data event control block (DECB) with the data necessary to access a general data set or a volume of a general data set.

The function opens or closes the general data set, establishing a correct CCHR address on the specified ECB data level or DECB for the indicated relative record number (RRN) and initializes areas of the ECB as required.

### Format

```
#include <tpfapi.h>
int      gdsnc(enum t_lvl level, unsigned char op, enum t_blktype size,
              int rrn_fmt, const char *dsn);
```

or

```
#include <tpfapi.h>
int      gdsnc(TPF_DECB *decb, unsigned char op, enum t_blktype size,
              int rrn_fmt, const char *dsn);
```

#### level

One of 16 possible values representing a valid ECB data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter specifies the extended file address reference word (FAXW) into which the CCHR address corresponding to the indicated RRN will be placed. Once returned by this function, the contents of this ECB data level should not be changed by the application. The CE1FMx field for the level specified should be initialized to zero or to the volume sequence number desired.

#### decb

A pointer to a DECB. This parameter specifies the FAXW into which the CCHR address corresponding to the indicated RRN will be placed. Once returned by this function, the contents of this DECB must not be changed by the application. The IDECFM0 field for the DECB specified must be initialized to zero or to the volume sequence number desired.

**op** Whether the data set is to be opened or closed. Use defined terms **GDSNC\_OPEN** or **GDSNC\_CLOSE**.

#### size

The size of the record. This parameter must belong to enumeration type `t_blktype`, defined in `tpfapi.h`. Use **L1**, **L2**, or **L4** for TPF formatted records, or **UNDEF** for MVS formatted records.

#### rrn\_fmt

Whether or not the data set being referred to has TPF format relative record numbers. Use the defined terms **GDSNC\_TPF\_FMT** to indicate TPF format, or **GDSNC\_NOT\_TPF\_FMT** to indicate MVS (or other) RRN formatting.

#### dsn

This parameter is treated as a pointer to type `char`, in which the data set name must appear. This data set name must be left-justified in an array of type `char`, padded with blanks, and must be 16 bytes long.

**Note:** Specifying **level**, **op**, **size**, or **rrn\_fmt** parameters that are not valid results in a system error with `exit`.

## Normal Return

Integer value of 0 indicating that the function completed successfully and that the FAXW has been updated with the proper CCHR address.

## Error Return

- 4 Invalid RRN for data set (**BAD\_DSN\_RRN**)
- 8 Invalid RRN for currently mounted volume (**BAD\_VOL\_RRN**)
- 12 Missing volume sequence number (**BAD\_VOL\_SEQ**)
- 16 Data set not mounted (**DSN\_NOT\_MNT**)
- 24 Device-dependent data table was not found (**DDT\_NOT\_FND**)

## Programming Considerations

- The CE1FMx field for the ECB data level or IDECFM0 field of the DECB specified must be initialized to zero or to the volume sequence number desired.
- The first fullword of the CE1FXx field of the ECB data level or IDECFX0 field of the DECB must contain zeros or the RRN of a desired record in a data set. It must be zero to access the first record in the data set.
- Except for the ID and record code check (RCC) portions of the file address reference words, the data inserted into the FARW by the TPF system must not be changed.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example opens a data set on level D9 using TPF system format, 1055-byte records, data set name VM1.KMV.FILE, and issues an appropriate message if there is an error.

```
#include <tpfapi.h>
int zero = 0;
:
ecbptr()->celfm9 = zero;
memcpy(&(ecbptr->celfx9[0]),&zero,4);
switch (gdsnc(D9,GDSNC_OPEN,L2,GDSNC_TPF_FMT,"VM1.KMV.FILE  "))
{
    case 0: break;
    case 4:
        /*          in TARGET(TPF) this call was:
           errno = 0x123;
           perror("BAD DSN RRN");
           abort();
           in ISO-C it looks like:
        */
        serrcop(SERRC_EXIT,0x123,"BAD DSN RRN",NULL) ;
    case 8: serrcop(SERRC_EXIT,0x234,"BAD VOL RRN",NULL) ;
    case 12: serrcop(SERRC_EXIT,0x345,"BAD VOL SEQ",NULL) ;
    case 16: serrcop(SERRC_EXIT,0x456,"DSN NOT MNT",NULL) ;
    case 24: serrcop(SERRC_EXIT,0x678,"DDT NOT FND",NULL) ;
}
```



## Related Information

- “gdsrsc—Get General Data Set Record” on page 244
- “raisa—General File Get File Address” on page 404.

See *TPF Application Programming* for more information about DECBs.

## gdsrsc—Get General Data Set Record

This function is provided to initialize a user's file address reference word with the data necessary to access a general data set or a volume of a general data set.

The function establishes a correct CCHR address on the specified entry control block (ECB) data level or data event control block (DECB) for the indicated relative record number (RRN) and initializes areas of the ECB as required. This function works on an open general data set.

### Format

```
#include <tpfapi.h>
int gdsrsc(TPF_DECB *decb, enum t_blktype size,
           const char *dsn);
```

or

```
#include <tpfapi.h>
int gdsrsc(TPF_DECB *decb, enum t_blktype size,
           const char *dsn);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter specifies the file address extension word (FAXW) onto which the CCHR address of the indicated RRN is to be placed. Once returned by this function, the contents of the file address reference word (FARW) and FAXW should not be altered by the application.

#### decb

A pointer to a DECB. This parameter specifies the FAXW into which the CCHR address of the indicated RRN will be placed. Once returned by this function, the contents of the FARW and FAXW must not be changed by the application.

#### size

The size of the record. This parameter must belong to the enumeration type `t_blktype`, defined in `tpfapi.h`. Use **L1**, **L2**, or **L4** for TPF-formatted records, or **UNDEF** for MVS-formatted records.

#### dsn

This parameter is treated as a pointer to type `char`, in which the data set name must appear. This data set name must be left-justified in a blank-padded array of type `char`, and must be 16 bytes long.

**Note:** Specifying **level** or **size** parameters that are not valid results in a system error with `exit`.

### Normal Return

A return of 0 indicates that the function completed successfully and that the FAXW has been updated with the proper CCHR address.

### Error Return

- 4 Invalid RRN for data set (**BAD\_DSN\_RRN**)
- 8 Invalid RRN for currently mounted volume (**BAD\_VOL\_RRN**)
- 12 Missing volume sequence number (**BAD\_VOL\_SEQ**)
- 16 Data set not mounted (**DSN\_NOT\_MNT**)

24 The device-dependent table was not found (**DDT\_NOT\_FND**).

## Programming Considerations

- The first fullword of the CE1FXx field of the ECB data level or IDECFX0 field of the DECB must contain zeros or the RRN of a desired record in the data set. This value must be zero to access the first record in the data set.
- Except for the ID and RCC portions of the file address reference words, the data inserted into the FARW by the TPF system must not be changed.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example converts the relative record number shown into a CCHR address to access the third record in data set MAR.PNJ.FILE. The program issues an appropriate message if there is an error.

```
#include <tpfapi.h>
int rrn = 2;
:
memcpy(&(ecbptr()->celfx3[0]),&rrn,4);
switch (gdsrsc(D3,UNDEF,"MAR.PNJ.FILE  "))
{
    case 4:
        /*          in TARGET(TPF) this call was:
           errno = 0x123;
           perror("BAD DSN RRN");
           abort();
           in ISO-C it looks like:
        */
        serrcop(SERRC_EXIT,0x123,"BAD DSN RRN",NULL) ;
    case 8: serrcop(SERRC_EXIT,0x234,"BAD VOL RRN",NULL) ;
    case 12: serrcop(SERRC_EXIT,0x345,"BAD VOL SEQ",NULL) ;
    case 16: serrcop(SERRC_EXIT,0x456,"DSN NOT MNT",NULL) ;
    case 24: serrcop(SERRC_EXIT,0x678,"DDT NOT FND",NULL) ;
}
```

## Related Information

- “gdsnc–Get Data Set Entry” on page 241
- “raisa–General File Get File Address” on page 404.

See *TPF Application Programming* for more information about DECBs.

---

## getc, getchar—Read a Character from Input Stream

These functions read a character from an input stream.

### Format

```
#include <stdio.h>
int getc(FILE *stream);
int getchar(void);
```

#### **stream**

The input stream.

These functions read a single character from the current **stream** position and advances the **stream** position to the next character. The `getchar` function is identical to `getc(stdin)`.

The `getc` and `fgetc` functions are identical. However, `getc` and `getchar` functions are provided in a highly efficient macro form. For performance purposes, it is recommended that the macro forms be used rather than the functional forms or `fgetc`. By default, `stdio.h` provides the macro versions of these functions.

However, to get the functional forms, do one or more of the following:

- Specify `#undef`, for example, `#undef getc`
- Surround the function name by parentheses, for example, `(getc)(fp)`.

The `getc` and `getchar` functions have the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an EOF has been reached.

**Note:** Because the `getc` macro reevaluates its input argument more than once, you should never pass a stream argument that is an expression with side effects.

### Normal Return

The `getc` and `getchar` functions return the character read.

### Error Return

A returned value of EOF indicates either an error or an EOF condition has occurred. If a read error occurs, the error indicator is set. If an EOF is encountered, the EOF indicator is set.

Use `ferror` or `feof` functions to determine whether an error or an EOF condition occurred.

**Note:** EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

### Examples

The following example gets a line of input from the `stdin` stream. You can also use `getc(stdin)` instead of `getchar` in the `for` statement to get a line of input from `stdin`.

```

#include <stdio.h>

#define LINE 80

int main(void)
{
    char buffer[LINE+1];
    int i;
    int ch;

    /* Keep reading until either:
       1. the length of LINE is exceeded or
       2. the input character is EOF or
       3. the input character is a new-line character
    */

    for ( i = 0; ( i < LINE ) && (( ch = getchar()) &xc1m.= EOF) &&
          ( ch !='\n' ); ++i )
        buffer[i] = ch;

    buffer[i] = '\0'; /* a string should always end with '\0' *

    printf( "The string is %s\n", buffer );
}

```

## Related Information

- “fgetc—Read a Character” on page 146
- “gets—Obtain Input String” on page 273
- “putc, putchar—Write a Character” on page 400
- “ungetc—Push Character to Input Stream” on page 664.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## getcc—Obtain Working Storage Block

This function obtains and attaches a working storage block to the indicated entry control block (ECB) data level or data event control block (DECB).

### Format

This function has four formats depending on the type of parameters coded:

```
#include <tpfapi.h>
void *getcc(enum t_lvl level, enum t_getfmt format, spec1);
```

or

```
#include <tpfapi.h>
void *getcc(enum t_lvl level, enum t_getfmt format, spec1, spec2);
```

or

```
#include <tpfapi.h>
void *getcc(TPF_DECB *decb, enum t_getfmt format, spec1);
```

or

```
#include <tpfapi.h>
void *getcc(TPF_DECB *decb, enum t_getfmt format, spec1, spec2);
```

#### level

One of 16 possible values representing a valid ECB data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0-F). This parameter represents an available core block reference word (CBRW) on which the requested working storage block will be placed.

#### decb

A pointer to a DECB. This parameter represents an available CBRW on which the requested working storage block will be placed.

#### format

For this parameter you must specify at least one term of enumeration type `t_getfmt`, defined in `tpfapi.h`, to determine how storage is allocated. Additionally, you can code the terms `GETCC_FILL` and `GETCC_COMMON` or `GETCC_PROTECTED`. If you code more than one term for format, you must separate terms with a plus sign (+). See the examples that follow.

The required term can be one of the following:

#### GETCC\_TYPE

Use this term to specify a logical block type. Code the logical block as the third parameter (`spec1`). The record ID attribute table (RIAT) is not accessed for this request. Use one of these predefined terms:

**L0** 127 bytes

**L1** 381 bytes

**L2** 1055 bytes

**L4** 4095 bytes.

#### GETCC\_SIZE

Use this term to specify the number of bytes of storage you need. Specify the number of bytes you need as an integer and as the third parameter (`spec1`). The RIAT table is not accessed for this request. The smallest available block size that satisfies the request is used. Requests for working storage in excess of 4095 bytes result in a system error with exit.

**GETCC\_ATTR0,...GETCC\_ATTR9, GETCC\_PRIME,or GETCC\_OVERFLOW**

Use one of these terms to specify a 2-character record ID to be used to scan the RIAT table for pool-type attributes specified by the ID for the ATTRx definition. Specify the 2-character record ID as the third parameter (spec1). GETCC\_PRIME and GETCC\_OVERFLOW are still supported for migration purposes only. Their values correspond to GETCC\_ATTR0 and GETCC\_ATTR1, respectively.

In addition, you can code one or both of the following terms:

**GETCC\_FILL**

Code this term to initialize the allocated storage to a specified hex value. Specify the hex value as the fourth parameter (spec2). If GETCC\_FILL is not coded, the storage is not initialized.

**GETCC\_COMMON**

Code this term to obtain storage from the pool of shared main storage. If you do not code GETCC\_COMMON or GETCC\_PROTECTED, the storage is not allocated from shared main storage.

**GETCC\_PROTECTED**

Code this term to obtain storage from the pool of shared main storage. The storage is key protected. If you do not code GETCC\_COMMON or GETCC\_PROTECTED, the storage is not allocated from shared main storage.

**spec1**

The getcc function always requires a third parameter depending on what term is coded for format.

If GETCC\_TYPE is coded, spec1 must be a logical block type.

If GETCC\_SIZE is coded, spec1 must be an integer indicating the number of bytes of storage you need.

If one of GETCC\_ATTR0 through GETCC\_ATTR9, GETCC\_PRIME, or GETCC\_OVERFLOW is coded, spec1 must be a 2-character record ID in double quotes.

**spec2**

If GETCC\_FILL is coded, you must code a fourth parameter specifying the hex value to which you want storage initialized.

**Normal Return**

Pointer to the newly obtained working storage block.

**Error Return**

Not applicable.

**Programming Considerations**

- Only 1 storage block at a time can be obtained through this function.
- The indicated ECB data level or DECB must be unoccupied when this function is called.
- Working storage blocks should be released at the first possible opportunity in order to avoid storage depletion.
- If there is no working storage, an incorrect RIAT ID is specified, an incorrect block type is specified, or a working storage block is already attached at the specified ECB data level or DECB, the result is a system error with exit.

## getcc

- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example obtains 6 working storage blocks on levels D2, DC, DF, D7, D9, and D3.

- The first is a RIAT-specified block of attribute 0.
- The second is a block-type call.
- The third is a self-defining size call for storage from the pool of shared main storage.
- The fourth is block-type call initializing the block to blanks.
- The fifth is a block-type call for storage from the pool of shared main storage, initializing the block to blanks.
- The sixth is a block-type call for storage from the pool of shared main storage, setting the block to storage key protected and initializing the block to zeros.

```
#include <tpfapi.h>
#include <c$am0sg.h>
struct am0sg *amsgl;           /* pointers to message blocks */
char *work1, *work2, *work3, *work4, *work5;
:
amsgl = (struct am0sg *) getcc(D2, GETCC_ATTR0, "OM");
work1 = (char *) getcc(DC, GETCC_TYPE, L2); /* Attaches 1055-byte block */
work2 = (char *) getcc(DF, (enum t_getfmt) (GETCC_SIZE+GETCC_COMMON), 1100);
/* Attaches from shared storage a
4095 byte block since the requested
size will not fit in a 1055/L2 block */
work3 = (char *) getcc(D7, (enum t_getfmt) (GETCC_TYPE+GETCC_FILL), L2, ' ');
/* Attaches a 1055 byte block
initialized to blanks */
work4 = (char *) getcc(D9, (enum t_getfmt) (GETCC_TYPE+GETCC_FILL+GETCC_COMMON), L4, ' ');
/* Attaches a 4095 byte block
from the pool of shared
main storage and initializes
it to blanks */
work5 = (char *) getcc(D3, (enum t_getfmt) (GETCC_TYPE+GETCC_PROTECTED+GETCC_FILL), L4, 0x00);
/* Attaches a 4095 byte block
from the pool of shared main
storage, sets the block to
storage key protected and
initializes it to zeros */
```

The following example uses DECBs to get storage blocks instead of ECB data levels.

```
#include <tpfapi.h>
#include <c$am0sg.h>
struct am0sg *amsgl;           /* pointers to message blocks*/
TPF_DECB *decbl, *decbl2, *decbl3, *decbl4, *decbl5, *decbl6;
char *work1, *work2, *work3, *work4, *work5;
:
amsgl = (struct am0sg *) getcc(decbl, GETCC_ATTR0, "OM");
work1 = (char *)getcc(decbl2, GETCC_TYPE, L2); /* Attaches 1055-byte block */
work2 = (char *)getcc(decbl3, (enum t_getfmt) (GETCC_SIZE+GETCC_COMMON), 1100);
/* Attaches from shared storage a
4095 byte block since the requested
size will not fit in a 1055/L2 block */
work3 = (char *)getcc(decbl4, (enum t_getfmt) (GETCC_TYPE+GETCC_FILL), L2, ' ');
/* Attaches a 1055 byte block
initialized to blanks */
work4 = (char *)getcc(decbl5, (enum t_getfmt) (GETCC_TYPE+GETCC_FILL+GETCC_COMMON), L4, ' ');
/* Attaches a 4095 byte block
from the pool of shared
```



```
main storage and initializes  
it to blanks */  
work5 = (char *)getcc(dec6, (enum t_getfmt) (GETCC_TYPE+GETCC_PROTECTED+GETCC_FILL), L4, 0x00);  
/* Attaches a 4095 byte block  
from the pool of shared main  
storage, sets the block to  
storage key protected and  
initializes it to zeros */
```

## Related Information

- “detac\_id–Detach a Working Storage Block from the ECB” on page 88
- “relcc–Release Working Storage Block” on page 421.

See *TPF Application Programming* for more information about DECBs.

## getcwd–Get Path Name of the Working Directory

This function gets path name of the working directory.

### Format

```
#include <unistd.h>
char *getcwd(char *buffer, size_t size);
```

#### size

The number of characters in the **buffer** area.

#### buffer

The name of the buffer that will be used to hold the path name of the working directory. **buffer** must be big enough to hold the working directory name, plus a terminating NULL to mark the end of the name.

This function determines the path name of the working directory and stores it in **buffer**.

### Normal Return

If successful, the `getcwd` function returns a pointer to the buffer.

### Error Return

If unsuccessful, the `getcwd` function returns a NULL pointer.

If `getcwd` function fails, it sets `errno` to one of the following:

<b>EACCES</b>	The process did not have read or search permission on some component of the working directory's path name.
<b>EINVAL</b>	<code>size</code> is less than or equal to zero.
<b>EIO</b>	An input/output error occurred.
<b>ERANGE</b>	<code>size</code> is greater than zero, but less than the length of the working directory's path name, plus 1 for the terminating NULL.

### Programming Considerations

If **buffer** is a NULL pointer, the `getcwd` function allocates **size** bytes and returns the address of the allocated buffer. The caller should free the buffer (using the `free` function) when it is no longer needed. This behavior is provided for Berkeley Software Distribution (BSD) compatibility, but this use of the `getcwd` function is *not* recommended because the preferred usage is for the user to manage the buffer where output is returned.

### Examples

The following example determines the working directory.

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

main() {
    char cwd[256];

    if (chdir("/tmp") != 0)
        perror("chdir() error");
    else {
        if (getcwd(cwd, sizeof(cwd)) == NULL)
            perror("getcwd() error");
```

```
        else
            printf("current working directory is: %s\n", cwd);
    }
}
```

**Output**

current working directory is: /tmp

**Related Information**

“chdir—Change the Working Directory” on page 30.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## getegid—Get the Effective Group ID

This function gets the effective group ID (GID) of the calling function's process.

### Format

```
#include <unistd.h>
gid_t getegid(void);
```

### Normal Return

The `getegid` function returns the effective group ID. It is always successful.

### Error Return

Not applicable.

## Programming Considerations

None.

## Examples

The following example provides information for the effective group ID of the caller:

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

int main(void) {

    printf("My group id is %d\n", (int) getegid() );
    return 0;
}
```

The example output from the `getegid` function obtains the effective group ID and places it in variable type `gid_t`.

My group id is 100

## Related Information

- “`getgid`—Get the Real Group ID” on page 260
- “`setegid`—Set the Effective Group ID” on page 462
- “`setgid`—Set the Group ID to a Specified Value” on page 468.

---

## getenv—Get Value of Environment Variables

This function searches the environment list for the variable and returns a pointer to a string containing its associated value.

### Format

```
#include <stdlib.h>
char *getenv(const char *varname);
```

**varname**

A variable in the environment list.

### Normal Return

Pointer to a string containing the current value associated with the variable name.

### Error Return

If the variable name is not found, the getenv function returns a null pointer.

## Programming Considerations

You **must** copy the string that is returned because a subsequent call to getenv could overwrite it.

### Examples

The following example prints the value of the PATH environment variable pointed to by \*pathvar.

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

int main(void)
{
    const char *pathvar = getenv("PATH");
    printf("PATH = %s\n", pathvar ? pathvar : "<NULL>");
}
```

### Related Information

- “setenv—Add, Change, or Delete an Environment Variable” on page 463
- “unsetenv—Delete an Environment Variable” on page 671.

## geteuid—Get the Effective User ID

This function gets the effective user ID (UID) of the calling function's process.

### Format

```
#include <unistd.h>
uid_t geteuid(void);
```

### Normal Return

The `geteuid` function returns the effective user ID of the calling process. It is always successful.

### Error Return

Not applicable.

## Programming Considerations

None.

## Examples

The following example provides information for the effective user ID of the caller:

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

int main(void) {

    char prthdr[] =
        "getpwuid() returned the following info for your user id:"
    struct passwd *p;
    uid_t uid;

    if ( ( p = getpwuid ( uid = geteuid() ) ) == NULL )
        perror ( "getpwuid() error" );
    else {
        printf( "%s\n", prthdr );
        printf( " pw_name  : %s\n", p->pw_name );
        printf( " pw_uid   : %d\n", p->pw_uid );
        printf( " pw_gid   : %d\n", p->pw_gid );
        printf( " pw_dir   : %s\n", p->pw_dir );
        printf( " pw_shell : %s\n", p->pw_shell);
    }
    return 0;
}
```

The `geteuid` function returns the following information for the user ID of the calling function:

```
getpwuid() returned the following info for your user id:
pwname   : TPFUSER1
pw_uid   : 260
pw_gid   : 100
pw_dir   :
pw_shell :
```

## Related Information

- “`getpwuid`—Access the User Database by User ID” on page 271
- “`getuid`—Get the Real User ID” on page 276
- “`seteuid`—Set the Effective User ID” on page 466

- “setuid–Set the Real User ID” on page 472.

## getfc—Obtain File Pool Address

This function obtains a file address (and optionally, a working storage block) based on attributes associated with a record ID.

### Format

```
#include    <tpfio.h>
unsigned int getfc(enum t_lvl level, int type, const char *id, int block,
                  int error, ... );
```

or

```
#include    <tpfio.h>
TPF_FA8    getfc(TPF_DECB *decb, int type, const char *id, int block,
                  int error, ...);
```

#### level

One of 16 possible values representing a valid entry control block (ECB) data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter represents an available file address reference word (FARW) location where the file address will be placed.

#### decb

A pointer to a data event control block (DECB). This parameter represents an available FARW location where the file address will be placed.

#### type

Specify a pool type attribute with the defined terms **GETFC\_TYPE0** through **GETFC\_TYPE9** associated with the `id` in the record ID attribute table (RIAT). The defined terms **GETFC\_PRIME** and **GETFC\_OVERFLOW** are still supported for migration purposes only. **GETFC\_PRIME** corresponds to **GETFC\_TYPE0** and **GETFC\_OVERFLOW** corresponds to **GETFC\_TYPE1**.

**id** A pointer to a 2-character string bearing the record ID for which this file allocation is to take place. This ID is used to scan the RIAT table for a match to determine record size and pool type.

#### block

Whether or not a working storage block is to be simultaneously obtained whose characteristics are determined by the RIAT table. Code **GETFC\_BLOCK** to obtain a block and a file address, or **GETFC\_NOBLOCK** to obtain only a file address.

Additionally, when **GETFC\_BLOCK** is specified, you can code one or both of the terms **GETFC\_COMM** and **GETFC\_FILL**. If you code more than one term for `block`, you must separate them with a logical OR (`|`) sign. (See the example that follows.) **GETFC\_COMM** indicates that a common block is to be acquired for the **GETFC\_BLOCK** option. **GETFC\_FILL** along with **GETFC\_BLOCK** indicates that the block acquired should be initialized with the fill character, **FILLC**, specified as the first optional parameter. **GETFC\_NOCOMM** and **GETFC\_NOFILL**, meaning not common block and no fill, respectively, are the defaults and need not be coded.

#### error

Whether or not control is to be returned to the operational program in the event of an error. Code **GETFC\_SERRC** to transfer control to the system error routine (with `exit`) if the file or core storage cannot be obtained, as requested. Code **GETFC\_NOSERRC** to have control returned to the caller.



... The character used to initialize the block if **GETFC\_BLOCK** and **GETFC\_FILL** are both specified. (The fill character may be specified in hexadecimal.)

## Normal Return

Unsigned integer value representing a file address, or an 8-byte file address defined as type `TPF_FA8`.

## Error Return

Integer value of zero if **GETFC\_NOSERRC** is coded; otherwise, loss of control and system error with exit.

## Programming Considerations

- The record ID specified as parameter **id** must exist in the RIAT table.
- The CBRW specified by the **level** or the **decb** parameter must be unoccupied if **GETFC\_BLOCK** is coded.
- Use of this function may result in the equivalent of a waitc.
- File pool addresses that are acquired in a global transaction (that is, after a `tx_begin` function call but before a `tx_commit` or `tx_rollback` function call) will be released if the transaction is rolled back using a `tx_rollback` function call.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example obtains file storage for a message block on level D2 and obtains a working storage block of the appropriate size on the same level.

```
#include <tpfio.h>
#include <c$am0sg.h>
struct am0sg *amsg;           /* pointers to message blocks */
:
amsg = ecblptr()->celcr1;     /* Base prime message block */
if (!(amsg->am0fch = getfc(D2,GETFC_TYPE1,"OM",GETFC_BLOCK|GETFC_FILL,
                          GETFC_NOSERRC,' ')))
    exit(0x33001);           /* Dump w/exit if getfc() failed */
```

The following example obtains file storage for a message block on the DECB and obtains a working storage block of the appropriate size on the same DECB.

```
#include <tpfio.h>
#include <c$am0sg.h>
TPF_FA8 address;
TPF_DECB *decb;
:
if (!(address = getfc(decb,GETFC_TYPE1,
                     "OM",(int)(GETFC_BLOCK+GETFC_FILL),
                     GETFC_NOSERRC,' ')))
    exit(0x33001);           /* Dump w/exit if getfc failed */
```

## Related Information

- “relfc—Release File Pool Storage” on page 423
- “waitc—Wait For Outstanding I/O Completion” on page 686.

See *TPF Application Programming* for more information about DECBs.

## getgid—Get the Real Group ID

This function gets the real group ID (GID) of the calling function's process.

### Format

```
#include <unistd.h>
gid_t getgid(void);
```

### Normal Return

The `getgid` function returns the real group ID of the calling process. It is always successful.

### Error Return

Not applicable.

## Programming Considerations

None.

## Examples

The following example provides information for the group ID of the caller:

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

int main(void) {
    printf( "My Group ID is %d\n", (int) getgid() );
    return 0;
}
```

The `getgid` function returns the following information for the group ID of the calling function.

My Group ID is 100

## Related Information

- “`geteuid`—Get the Effective User ID” on page 256
- “`getgrgid`—Access the Group Database by ID” on page 261
- “`getuid`—Get the Real User ID” on page 276
- “`setgid`—Set the Group ID to a Specified Value” on page 468.

## getgrgid—Access the Group Database by ID

This function provides information about the group specified by the group ID (GID) and its members.

### Format

```
#include <grp.h>
struct group *getgrgid(gid_t gid);
```

#### **gid**

A group ID value.

### Normal Return

If successful, the `getgrgid` function returns a pointer to a group structure containing an entry from the group database with the specified **gid**. The return value is a pointer to static data that is overwritten by each call.

This group structure, defined in the `grp.h` header file, contains the following members:

#### **gr\_name**

Pointer to the group name character string.

#### **gr\_gid**

The numeric GID.

#### **gr\_mem**

A NULL-terminated array of pointers to the individual member names in this group.

### Error Return

If unsuccessful, the `getgrgid` function returns a NULL pointer and sets `errno` to one of the following:

#### **EIO**

An input/output (I/O) error occurred.

#### **EMFILE**

The process has already reached its maximum number of open file descriptors. This limit is given by `OPEN_MAX`, which is defined in the `limits.h` header file.

#### **ENFILE**

The TPF system has already reached its maximum number of open files.

### Programming Considerations

- This function performs I/O operations, which cause the entry control block (ECB) to give up control.
- If an I/O error occurs, an unsolicited error message is sent to a computer room agent set (CRAS).
- When an I/O error occurs, `errno` is set by the I/O operation that was being performed (`open`, `read`, `write`, or `close` function).

### Examples

The following example provides the root GID and group name.

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

## getgrgid

```
int main(void) {
    struct group *grp;
    struct stat info;

    if ( stat("/", &info ) < 0 )
        perror ( "stat() error" );
    else {
        printf( "The root is owned by gid %d\n", info.st_gid );
        if ( ( grp = getgrgid ( info.st_gid ) ) == NULL )
            perror ( "getgrgid() error" );
        else
            printf ( "This group name is %s\n", grp->gr_name );
    }
    return 0;
} /* end of main */
```

### Output

```
The root is owned by GID 0
This group name is root
```

## Related Information

- “getgid—Get the Real Group ID” on page 260
- “getgrnam—Access the Group Database by Name” on page 263.

## getgrnam—Access the Group Database by Name

This function accesses the group structure containing an entry from the group database with the specified **name**.

### Format

```
#include <grp.h>
struct group *getgrnam(const char *name);
```

#### **name**

A pointer to a character string of a group name (NULL-terminated).

### Normal Return

If successful, the getgrnam function returns a pointer to a group structure containing an entry from the group database with the specified **name**. The return value is a pointer to static data that is overwritten by each call.

This group structure, defined in the grp.h header file, contains the following members:

#### **gr\_name**

Pointer to the group name character string.

#### **gr\_gid**

The numeric group ID (GID).

#### **gr\_mem**

A NULL-terminated array of pointers to the individual member names in this group.

### Error Return

If unsuccessful, the getgrnam function returns a NULL pointer and sets errno to one of the following:

#### **EINVAL**

Indicates that the length of the **name** parameter is incorrect. The length of **name** must be equal to the length of the group name in the group file; it must also be less than 256 characters.

#### **EIO**

An input/output (I/O) error occurred.

#### **EMFILE**

The process has already reached its maximum number of open file descriptors. This limit is given by OPEN\_MAX, which is defined in the limits.h header file.

#### **ENFILE**

The TPF system has already reached its maximum number of open files.

### Programming Considerations

- This function performs I/O operations, which cause the entry control block (ECB) to give up control.
- If an I/O error occurs, an unsolicited error message is sent to a computer room agent set (CRAS).
- When an I/O error occurs, errno is set by the I/O operation that was being performed (open, read, write, or close function).

## getgrnam

### Examples

The following example provides the members of a group.

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

int main(void) {

    char  grpname[] = "users", **curr;

    struct group *grp;

    if ((grp = getgrnam(grpname)) == NULL)
        perror ( "getgrnam() error" );
    else {
        printf( "The following are members of group %s:\n", grpname );
        for (curr=grp->gr_mem; (*curr) != NULL; curr++ )
            printf("  %s\n", *curr);
    }
    return 0;
}
```

#### Output

The following members of group users:  
tpfuser1  
tpfuser2  
tpfuser3

### Related Information

- “getgrgid–Access the Group Database by ID” on page 261.

## getpc—Get Program and Lock in Core

This function is used to lock a program in core or locate a program either in core or on file.

### Format

```
#include <tpfapi.h>
int      getpc(const char *name, int lock, enum t_lvl level,
              void *core_addr, int error, int id,
              const char *loadset);
```

#### name

A pointer to the character representation of the program name. The name is 4 alphanumeric characters that should have been allocated at system generation time. The program must be nonprivate.

#### lock

This argument is treated as an integer describing whether the program should be locked in core. Use the defined terms **GETPC\_LOCK** to lock the program in core, **GETPC\_SPECIAL** to lock the program in core and mark it as special, or **GETPC\_NOLOCK** to locate a program but not lock it in core.

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This indicates which ECB FARW should be initialized with the program's file address and record ID. **NO\_LVL** can be specified if the file address is not required.

#### core\_addr

Receives the core address of the program.

#### error

This argument is treated as an integer describing whether control should be returned to the caller on an error or if an OPR-066 should be issued. Use the defined term **GETPC\_NODUMP** to denote that control should be returned to the caller, or **GETPC\_DUMP** to denote that the service routine should issue an OPR-066. The possible errors are: the program name could not be found, the program name was found but the loadset name could not be found, the program is allocated as private, or the program already has the special indicator set.

**id** Indicates which MDBF index will be used for running this function. Use the defined term **GETPC\_PBI** to denote that the program base ID should be used, or **GETPC\_DBI** to denote that the database ID should be used. **GETPC\_DBI** is not valid when coded with **GETPC\_LOCK**.

#### loadset

A pointer to a valid loadset name or BASE if referring to the base-allocated program. If this parameter is used, the active program that is associated with the specified loadset is affected. If this parameter is specified as NULL, the version of the program associated with the requesting ECB is affected.

### Normal Return

0

### Error Return

A nonzero error return code, as follows in Table 9.

## getpc

Table 9. *getpc* Error Return

Numbers	Return Code	Description
#define GETPC_SNAPC_ERR	4	
#define GETPC_LS_NF	8	LOADSET VERSION NOT FOUND
#define GETPC_SP_SET	16	SPECIAL LOCK NOT/ALREADY SET
#define GETPC_RETRIEVE_ERROR	32	PROGRAM RETRIEVAL ERROR
#define GETPC_PRIVATE	64	PROGRAM IS A PRIVATE PROGRAM
#define GETPC_PGM_NF	128	PROGRAM NOT FOUND

## Programming Considerations

- Only nonprivate programs can be locked in core.
- This function is not valid for C-type programs.
- Only 1 program at a time may be locked through this function. Separate requests must be made for each program required.
- When **GETPC\_NOLOCK** is specified, either the **level** parameter or the **core\_addr** parameter must be specified.
- When the program is no longer needed in core it should be unlocked using the **relpc** function.
- When **GETPC\_SPECIAL** is specified on the **lock** parameter of the **getpc** function, **RELPC\_SPECIAL** must be specified on the **unlock** parameter of the **relpc** function.

## Examples

The following example obtains the file address of program QZZ0 on data level A, and the program's core address also. The information is obtained using the PBI. If an error occurs, the service routine should issue an OPR-066.

```
#include <tpfapi.h>
unsigned int pgm_core_addr;
struct pat *base_pat;
:
:
getpc("QZZ0",GETPC_NOLOCK,DA,&pgm_core_addr,GETPC_DUMP,GETPC_PBI,
    NULL );
:
:
```

## Related Information

"relpc—Release Program from Core Lock" on page 425.



---

## getpid—Obtain a Process ID

This function permits a process to obtain its process identifier (ID).

### Format

```
#include <unistd.h>
pid_t getpid(void);
```

### Normal Return

The getpid function is always successful.

The getpid function returns the process ID of the calling process.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example shows how to print the current process ID.

```
#include <stdio.h>
#include <unistd.h>
...
printf("My PID = %d\n", getpid());
{
```

### Related Information

- “getppid—Obtain the Parent Process ID” on page 268
- “kill—Send a Signal to a Process” on page 298
- “tpf\_fork—Create a Child Process” on page 594.

---

### getppid—Obtain the Parent Process ID

This function permits a process to obtain the process identifier (ID) of its parent process.

#### Format

```
#include <unistd.h>
pid_t getppid(void);
```

#### Normal Return

The `getppid` function is always successful.

If the calling process was created by the `tpf_fork` function and the parent process still exists at the time of the `getppid` function call, this function returns the process ID of the parent process. Otherwise, this function returns a value of 1.

#### Error Return

Not applicable.

#### Programming Considerations

None.

#### Examples

The following example shows how to print the parent process ID.

```
#include <stdio.h>
#include <unistd.h>
...
printf("Parent PID = %d\n", getppid());
{
```

#### Related Information

- “`getpid`—Obtain a Process ID” on page 267
- “`kill`—Send a Signal to a Process” on page 298
- “`tpf_fork`—Create a Child Process” on page 594.

---

## getpwnam—Access the User Database by Name

This function gets information about the user with the specified **name**.

### Format

```
#include <pwd.h>
struct passwd *getpwnam(char const *name)
```

#### **name**

A pointer of type character to a string **name** (NULL-terminated) representing the name of the user ID (UID) to be retrieved from file. **name** must be lowercase.

### Normal Return

If successful, the getpwnam function returns a pointer to struct passwd with an entry from the user database for the specified **name**.

The return value is a pointer to static data that is overwritten by each call.

struct passwd, defined in the pwd.h C header file, contains the following members:

#### **pw\_name**

Pointer to the password user name character string.

#### **pw\_uid**

The numeric UID.

#### **pw\_gid**

The numeric group ID (GID).

#### **pw\_gecos**

Pointer to the comment character string.

#### **pw\_dir**

Pointer to the initial working directory character string.

#### **pw\_shell**

Pointer to the character string name of the initial shell (user program).

### Error Return

If unsuccessful, the getpwnam function returns a NULL pointer and sets errno to the appropriate value, as follows:

#### **EINVAL**

Indicates that the length of the **name** parameter is incorrect. The length of **name** must be equal to the length of the user name in the password user ID file; it must also be less than 256 characters.

#### **EIO**

An input/output (I/O) error occurred.

#### **EMFILE**

The process has already reached its maximum number of open file descriptors. This limit is given by OPEN\_MAX, which is defined in the limits.h header file.

#### **ENFILE**

The TPF system has already reached its maximum number of open files.

## getpwnam

### Programming Considerations

- This function performs I/O operations, which cause the entry control block (ECB) to give up control.
- If an I/O error occurs, an unsolicited error message is sent to a computer room agent set (CRAS).
- When an I/O error occurs, `errno` is set by the I/O operation that was being performed (open, read, write, or close function).

### Examples

The following example provides information for user name **tpfuser1**.

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

int main(void) {
    char prthdr[] =
        "getpwnam() returned the following info for your user id:"
    struct passwd *p;
    char user[] = "tpfuser1";
    if ( ( p = getpwnam ( user ) ) == NULL )
        perror ( "getpwnam() error" );
    else {
        printf( "%s\n", prthdr );
        printf( " pw_name  : %s\n", p->pw_name );
        printf( " pw_uid   : %d\n", (int) p->pw_uid );
        printf( " pw_gid   : %d\n", (int) p->pw_gid );
        printf( " pw_dir   : %s\n", p->pw_dir );
        printf( " pw_shell : %s\n", p->pw_shell );
    }
    return 0;
}
```

The `getpwnam` function returns the following information for the user ID of the calling function.

```
getpwnam() returned the following info for your user id:
pwname   : tpfuser1
pw_uid   : 260
pw_gid   : 100
pw_dir   : /usr
pw_shell :
```

### Related Information

“`getpwuid`—Access the User Database by User ID” on page 271.

## getpwuid—Access the User Database by User ID

This function gets information about the user with the specified **uid**.

### Format

```
#include <pwd.h>
struct passwd *getpwuid(uid_t uid)
```

**uid**

A UID value representing the user ID to be retrieved from file.

### Normal Return

If successful, the `getpwuid` function returns a pointer to `struct passwd` containing an entry from the user database for the specified **uid**. The return value is a pointer to static data that is overwritten by each call.

`struct passwd`, defined in the `pwd.h` C header file, contains the following members:

**pw\_name**

Pointer to the password user name character string.

**pw\_uid**

The numeric user ID (UID).

**pw\_gid**

The numeric group ID (GID).

**pw\_gecos**

Pointer to the comment character string.

**pw\_dir**

Pointer to the initial working directory character string.

**pw\_shell**

Pointer to the character string name of the initial shell (user program).

### Error Return

If unsuccessful, the `getpwuid` function returns a NULL pointer and sets `errno` to one of the following:

**EIO**

An input/output (I/O) error occurred.

**EMFILE**

The process has already reached its maximum number of open file descriptors. This limit is given by `OPEN_MAX`, which is defined in the `limits.h` header file.

**ENFILE**

The TPF system has already reached its maximum number of open files.

### Programming Considerations

- This function performs I/O operations, which cause the entry control block (ECB) to give up control.
- If an I/O error occurs, an unsolicited error message is sent to a computer room agent set (CRAS).
- When an I/O error occurs, `errno` is set by the I/O operation that was being performed (`open`, `read`, `write`, or `close` function).

## getpwuid

### Examples

The following example shows accessing the database for user ID 0.

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

int main(void) {

    char prthdr[] =
        "getpwuid() returned the following info for your user id:"
    struct passwd *p;
    uid_t uid = 0;

    if ( ( p = getpwuid ( uid ) ) == NULL )

        perror ( "getpwuid() error" );
    else {
        printf( "%s\n", prthdr );
        printf( " pw_name : %s\n", p->pw_name );
        printf( " pw_uid  : %d\n", (int) p->pw_uid );
        printf( " pw_gid  : %d\n", (int) p->pw_gid );
        printf( " pw_dir  : %s\n", p->pw_dir );
        printf( " pw_shell : %s\n", p->pw_shell);
    }
    return 0;
}
```

The getpwuid function returns the following information for the user ID of the calling function.

```
getpwuid() returned the following info for your user id:
pwname   : root
pw_uid   : 0
pw_gid   : 0
pw_dir   : /
pw_shell :
```

### Related Information

“getuid—Get the Real User ID” on page 276.

---

## gets—Obtain Input String

This function reads a string from the standard input stream.

### Format

```
#include <stdio.h>
char *gets(char *buffer);
```

#### **buffer**

This argument is a pointer to available working storage of sufficient size to hold the expected input.

This function reads bytes from the standard input stream `stdin`, and stores them in the array pointed to by **buffer**. The line consists of all characters up to and including the first new-line character (`\n`) or end-of-file (EOF). The `gets` function discards any new-line character, and the null character (`\0`) is placed immediately after the last byte read. If there is an error, the value stored in **buffer** is undefined.

The `gets` function has the same restriction as any read operation, such as a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an EOF has been reached.

#### **TARGET(TPF) restrictions**

The `gets` function in the TARGET(TPF) library is a limited and non-standard substitute for the standard `gets` function. The TARGET(TPF) library has no support for files, `stdin`, or file redirection.

The TARGET(TPF) version of the `gets` function does **not** have the same effect as the ISO-C version. Use of both versions in the same application can have unpredictable results.

### Normal Return

If successful, the `gets` function returns its argument.

### Error Return

A NULL pointer returned value indicates an error or an EOF condition with no characters read.

Use `ferror` or `feof` functions to determine which of these conditions occurred. Note that EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does **not** turn on the EOF indicator.

## Programming Considerations

### TARGET(TPF) restrictions

- Owing to the large diversity of terminal devices and network support in the TPF system, your installation must code this function to work with its requirements. All programming considerations related to the use of this function, therefore, depend on the manner of implementation.
- A valid input message block must exist in a location established by your installation. If the block cannot be found or bears an improper record ID, zero is returned.
- This function returns the address of a line that originally ended with a new-line character (\n). This character is replaced with a hex zero (\0) so that the returned line may be treated as a character string.
- Subsequent calls to gets function will return the beginning address of the next new-line-terminated string, if any, in the message. When no more input exists, a null pointer is returned.

## Examples

The following example gets a line of input from stdin.

```
#include <stdio.h>
#define MAX_LINE 100

int main(void)
{
    char line[MAX_LINE];
    char *result;

    printf("Enter string:\n");
    if ((result = gets(line)) != NULL)
        printf("string is %s\n", result);
    else
        if (ferror(stdin))
            printf("Error\n");
}
```

## Related Information

- “feof—Test End-of-File Indicator” on page 141
- “ferror—Test for Read/Write Errors” on page 143
- “fgets—Read a String from a Stream” on page 150
- “fputs—Write a String” on page 211.



---

## gettimeofday–Get Time

The gettimeofday function obtains the current time, expressed as seconds and microseconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, and stores it in a timeval structure.

### Format

```
#include <sys/time.h>
int gettimeofday(struct timeval *tp, struct timezone *tzp);
```

#### **sys/time.h**

The sys/time.h header file is used instead of the sys\_time.h or sys/time.h header files.

**Note:** See *OS/390 C/C++ Run-Time Library Reference* for more information about the gettimeofday function.

## getuid—Get the Real User ID

This function finds the real user ID (UID) of the calling function's process.

### Format

```
#include <unistd.h>
uid_t getuid(void);
```

### Normal Return

The `getuid` function returns the real user ID of the calling function's process. It is always successful.

### Error Return

Not applicable.

## Programming Considerations

None.

## Examples

The following example provides information for the user ID of the caller.

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

int main(void) {
    printf ( "My real user id is %d\n", (int) getuid() );
    return 0;
}
```

The `getuid` function returns the following information for the user ID of the calling function:

My real user id is 500

## Related Information

- “`getpwuid`—Access the User Database by User ID” on page 271
- “`seteuid`—Set the Effective User ID” on page 466
- “`setuid`—Set the Real User ID” on page 472.

## glob—Address TPF Global Field or Record

This function is used to return a pointer to a global field or a record's global directory entry named in `c$globz.h`. Use of a term not defined in `c$globz.h` will have unpredictable results. This function provides read-only reference to the specified global. To modify global fields, use the `glob_modify`, `glob_update`, or `global` function.

### glob—Generate Path Name

The standard C library function `glob` is supported but not documented in this book; however, it is documented in *OS/390 C/C++ Run-Time Library Reference*.

## Format

```
#include <tpfglbl.h>
#include <c$globz.h>
void      *glob(unsigned int tagname);
```

### tagname

This argument, which must be defined in `c$globz.h`, is treated as an unsigned 32-bit integer that describes the displacement, length, and attributes of the global field or record.

## Normal Return

A global address. If the `tagname` parameter describes a global field, the address of the field is returned. If the `tagname` parameter describes a global record, the address of the global directory for that record is returned. A global directory address can point to either the main storage address or the file address of the record, depending on how globals are defined in your system. If the `tagname` describes the main storage address directory for a global record, an additional level of indirection is required to address the global record itself.

## Error Return

Not applicable.

## Programming Considerations

None.

## Examples

In the following example, the `glob` function returns the address of a 4 byte global field (which will be treated as a long int).

```
#include <tpfglbl.h>
#include <c$globz.h>

long int *longglob_ptr = glob(_lflld);
```

In the following example, the `glob` function returns the address of the global directory core address field for a global record. The global record itself is a struct `msgexp`.

## glob

```
#include <tpfglbl.h>
#include <c$globz.h>

struct msgexp **q05met_directory_ptr = glob(_q05met);
struct msgexp *q05met_ptr = *q05met_directory_ptr;
```

## Related Information

- “glob\_modify—Modify TPF Global Field or Record” on page 282
- “glob\_sync—Synchronize TPF Global Field or Record” on page 284
- “glob\_unlock—Unlock TPF Global Field or Record” on page 286
- “glob\_update—Update TPF Global Field or Record” on page 288
- “global—Operate on TPF Global Field” on page 290.

## glob\_keypoint—Keypoint TPF Global Field or Record

This function causes the keypointing of the specified TPF global field or record.

### Format

```
#include <tpfglbl.h>
#include <c$globz.h>
void glob_keypoint(unsigned int tagname);
```

#### tagname

This argument, which is defined in header file c\$globz.h, uniquely identifies the TPF global field or record to be keypointed.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The argument coded as **tagname** must be defined in header file c\$globz.h. Results will be unpredictable if this restriction is not heeded.
- This function performs the equivalent of a GLOUC assembler macro. If the tagname specifies a global record, that record is keypointed; if the tagname specifies a global field, the global record which contains that field is keypointed.

## Examples

The following example modifies and keypoints a global record.

```
#include <tpfglbl.h>
#include <c$globz.h>
...
{
    /******
    /* Increment data element mykdata in keypointable global record */
    /* _mykglob. */
    /******
    struct mykglbrec **mkgrrptr = glob(_mykglob);
    struct mykglbrec *mkgrrptr = *mkgrrptr;
    long newdata = mkgrrptr->mykdata + 1;
    glob_modify(_mykglob, &mkgrrptr->mykdata, &newdata,
               sizeof mkgrrptr->mykdata);
    glob_keypoint(_mykglob);
}
```

## Related Information

- “glob\_lock—Lock and Access Synchronizable TPF Global Field or Record” on page 280
- “glob\_modify—Modify TPF Global Field or Record” on page 282
- “glob\_sync—Synchronize TPF Global Field or Record” on page 284
- “glob\_unlock—Unlock TPF Global Field or Record” on page 286
- “glob\_update—Update TPF Global Field or Record” on page 288
- “glob—Address TPF Global Field or Record” on page 277.

## glob\_lock—Lock and Access Synchronizable TPF Global Field or Record

This function reserves exclusive write access to the specified TPF global field or record and forces refreshing of the core copy from the most current file copy. You should ensure that a lock is held for the shortest possible time because severe system degradation may result from other ECBs waiting for locks. This function is valid only for synchronizable TPF global fields and records.

### Format

```
#include <tpfglbl.h>
#include <c$globz.h>
void *glob_lock(unsigned int tagname);
```

#### tagname

This argument, which is defined in header file c\$globz.h, uniquely identifies the TPF global field or record to be locked.

### Normal Return

This function returns a pointer to the specified TPF global field or record's directory entry (for example, a pointer to the field or record's address).

### Error Return

Not applicable.

## Programming Considerations

- The argument coded as **tagname** must be defined in header file c\$globz.h. Results will be unpredictable if this restriction is not heeded.
- Programs using this function should be prepared to call glob\_update, glob\_sync or glob\_unlock as soon as possible to prevent severe system degradation caused by other ECBs waiting for the lock.
- Programs using this function should not have any pending I/O operations outstanding because this function may perform the equivalent of a waitc function.
- This function performs the equivalent of the SYNCC macro. See *TPF General Macros* for more information about the SYNCC macro.

## Examples

The following example locks and updates a synchronizable global field.

```
#include <tpfglbl.h>
#include <c$globz.h>
...
{
    /******
    /* Increment data element mysdata in synchronizable global record */
    /* _mysglob. */
    /******
    struct msglbrecc **msggrp_ptr = glob_lock(_mysglob);
    struct msglbrecc *msggrp_ptr = *msggrp_ptr;
    long newdata = msggrp_ptr->mysdata + 1;
    glob_update(_mysglob, &msggrp_ptr->mysdata, &newdata,
               sizeof msggrp_ptr->mysdata);
}
```

## **Related Information**

- “glob\_keypoint–Keypoint TPF Global Field or Record” on page 279
- “glob\_modify–Modify TPF Global Field or Record” on page 282
- “glob\_sync–Synchronize TPF Global Field or Record” on page 284
- “glob\_unlock–Unlock TPF Global Field or Record” on page 286
- “glob\_update–Update TPF Global Field or Record” on page 288
- “glob–Address TPF Global Field or Record” on page 277.

## glob\_modify—Modify TPF Global Field or Record

This function copies the modification data to the specified TPF global field or record.

### Format

```
#include <tpfglbl.h>
#include <c$globz.h>
int glob_modify(unsigned int tagname, void *dstptr,
                const void *srcptr, int length);
```

#### **tagname**

This argument, which is defined in header file `c$globz.h`, uniquely identifies the TPF global field or record to be accessed.

#### **dstptr**

The address of the data in the global field or record to be modified.

#### **srcptr**

The address of the data used to modify the global field or record.

#### **length**

The length of the modified data.

### Normal Return

#### **GLOB\_RC\_OK**

The global field or record was updated successfully.

### Error Return

#### **GLOB\_RC\_LENGTH\_ERROR**

The length parameter was not a positive integer, or would have extended the overlay past the end of the global being modified (as specified by the tagname parameter).

#### **GLOB\_RC\_RANGE\_ERROR**

The dstptr parameter was outside the range of addresses containing the global area as specified by the tagname parameter.

#### **GLOB\_RC\_DIRECTORY\_ERROR**

The global record directory field specified by the tagname parameter is a file address. (This return applies only to global records.)

#### **GLOB\_RC\_KEY\_ERROR**

The storage protection KEY for the global record being modified does not match the KEY for either GLOBAL1, GLOBAL2, or GLOBAL3. (This return applies only to global records.)

#### **GLOB\_RC\_OVERLAP\_ERROR**

The global area to be modified and the modification data overlap.

### Programming Considerations

- The argument coded as **tagname** must be defined in header file `c$globz.h`. Results will be unpredictable if this restriction is not heeded.
- A program should not give up control (by calling an I/O function, such as `finwc`) between accessing and updating a TPF global field or record. If it does, the data that the program accesses may have already been updated by another program by the time the program decides to update it. Use of the locking facility (see



“glob\_lock—Lock and Access Synchronizable TPF Global Field or Record” on page 280 ) provides additional security when updating synchronizable global fields.

- No additional functions are performed against the global field or record (keypointing, processor synchronization, and others). To perform additional functions, you must either use the glob\_update function, or subsequently call glob\_keypoint or glob\_sync.
- If the global field or record can be synchronized, glob\_lock must be called before calling glob\_modify.
- This function gets the appropriate global PSW key before updating the global area specified by tagname, and returns the PSW key to the working storage key before returning to the caller.

## Examples

The following example modifies a synchronizable global record.

```
#include <tpfglbl.h>
#include <c$globz.h>
...
{
    /******
    /* Increment data element mysdata in synchronizable global record */
    /* _mysglob. */
    /******
    struct mysglbrec **msgrrptr = glob_lock(_mysglob);
    struct mysglbrec *msgrrptr = *msgrrptr;
    long newdata = msgrrptr->mysdata + 1;
    glob_modify(_mysglob, &msgrrptr->mysdata, &newdata,
               sizeof msgrrptr->mysdata);
    glob_sync(_mysglob);
}
```

## Related Information

- “glob\_keypoint—Keypoint TPF Global Field or Record” on page 279
- “glob\_lock—Lock and Access Synchronizable TPF Global Field or Record” on page 280
- “glob\_sync—Synchronize TPF Global Field or Record” on page 284
- “glob\_unlock—Unlock TPF Global Field or Record” on page 286
- “glob\_update—Update TPF Global Field or Record” on page 288
- “glob—Address TPF Global Field or Record” on page 277.

## glob\_sync—Synchronize TPF Global Field or Record

This function synchronizes the field or record across processors in a loosely coupled complex and then unlocks the field or record. The `glob_lock` function must have previously been called. This function is valid only for synchronizable TPF global fields and records.

### Format

```
#include <tpfglbl.h>
#include <c$globz.h>
void glob_sync(unsigned int tagname);
```

#### tagname

This argument, which is defined in header file `c$globz.h`, uniquely identifies the TPF global field or record to be synchronized.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The argument coded as **tagname** must be defined in header file `c$globz.h`. Results will be unpredictable if this restriction is not heeded.
- Programs using this function should not have any pending I/O operations outstanding because this function may perform the equivalent of a `waitc` function.
- This function performs the equivalent of a `SYNCC SYNC` assembler macro.

## Examples

The following example locks, updates and synchronizes a global field.

```
#include <tpfglbl.h>
#include <c$globz.h>
...
{
    /******
    /* Increment data element mysdata in synchronizable global record */
    /* _mysglob. */
    /******
    struct mysglbrec **msggrp_ptr = glob_lock(_mysglob);
    struct mysglbrec *msggrp_ptr = *msggrp_ptr;
    long newdata = msggrp_ptr->mysdata + 1;
    glob_modify(_mysglob, &msggrp_ptr->mysdata, &newdata,
               sizeof msggrp_ptr->mysdata);
    glob_sync(_mysglob);
}
```

## Related Information

- “glob\_keypoint—Keypoint TPF Global Field or Record” on page 279
- “glob\_lock—Lock and Access Synchronizable TPF Global Field or Record” on page 280
- “glob\_modify—Modify TPF Global Field or Record” on page 282
- “glob\_unlock—Unlock TPF Global Field or Record” on page 286
- “glob\_update—Update TPF Global Field or Record” on page 288

- “glob—Address TPF Global Field or Record” on page 277.

### glob\_unlock–Unlock TPF Global Field or Record

This function releases the exclusive use of the specified TPF global field or record. The `glob_lock` function must have previously been called. This function is valid only for synchronizable TPF global fields and records.

#### Format

```
#include <tpfglbl.h>
#include <c$globz.h>
void glob_unlock(unsigned int tagname);
```

##### tagname

This argument, which is defined in header file `c$globz.h`, uniquely identifies the TPF global field or record to be unlocked.

#### Normal Return

Void.

#### Error Return

Not applicable.

### Programming Considerations

- The argument coded as **tagname** must be defined in header file `c$globz.h`. Results will be unpredictable if this restriction is not heeded.
- Programs using this function should not have any pending I/O operations outstanding because this function may perform the equivalent of a `waitc` function.
- A CTL-6A7 will result if the global field or record is not locked when this function is called.
- This function performs the equivalent of a SYNCC UNLOCK assembler macro.

### Examples

The following example locks, tests, and unlocks a synchronizable global field.

```
#include <tpfglbl.h>
#include <c$globz.h>
...
{
    /* *****
    /* Test data element mysdata in synchronizable global record      */
    /* _mysglob.                                                         */
    /* *****
    struct mysglbrec **msgrpтрptr = glob_lock(_mysglob);
    struct mysglbrec *msgrpтр = *msgrpтрptr;
    if (msgrpтр->mysdata != 0) { ... }
    glob_unlock(_mysglob);
}
```

### Related Information

- “glob\_keypoint–Keypoint TPF Global Field or Record” on page 279
- “glob\_lock–Lock and Access Synchronizable TPF Global Field or Record” on page 280
- “glob\_modify–Modify TPF Global Field or Record” on page 282
- “glob\_sync–Synchronize TPF Global Field or Record” on page 284
- “glob\_update–Update TPF Global Field or Record” on page 288

- “glob—Address TPF Global Field or Record” on page 277.

## glob\_update—Update TPF Global Field or Record

This function updates data contained in the specified TPF global field or record. All functions requested to be performed against the TPF global field or record (keypointing, processor synchronization, and others) are validated based on the definition of the tagname. If the TPF global field or record is synchronizable, glob\_lock must be called before glob\_update.

### Format

```
#include <tpfglbl.h>
#include <c$globz.h>
int glob_update(unsigned int tagname, void *dstptr,
               const void *srcptr, int length);
```

#### tagname

This argument, which is defined in header file c\$globz.h, uniquely identifies the TPF global field or record to be accessed.

#### dstptr

The address of the data in the global field or record to be updated.

#### srcptr

The address of the data used to update the global field or record.

#### length

The length of the updated data.

### Normal Return

#### GLOB\_RC\_OK

The global field or record was updated successfully. If the global area is synchronizable it will be synchronized; if the global area is keypointable it will be keypointed.

### Error Return

#### GLOB\_RC\_LENGTH\_ERROR

The length parameter was not a positive integer, or would have extended the overlay past the end of the global being modified (as specified by the tagname parameter).

#### GLOB\_RC\_RANGE\_ERROR

The dstptr parameter was outside the range of addresses containing the global area as specified by the tagname parameter.

#### GLOB\_RC\_DIRECTORY\_ERROR

The global record directory field specified by the tagname parameter is a file address. (This return applies only to global records.)

#### GLOB\_RC\_KEY\_ERROR

The storage protection KEY for the global record being modified does not match the KEY for either GLOBAL1, GLOBAL2, or GLOBAL3. (This return applies only to global records.)

#### GLOB\_RC\_OVERLAP\_ERROR

The global area to be modified and the modification data overlap.

### Programming Considerations

- The argument coded as **tagname** must be defined in header file c\$globz.h. Results will be unpredictable if this restriction is not heeded.

- Programs using this function should not have any pending I/O operations outstanding because this function may perform the equivalent of a `waitc` function.
- This function gets the appropriate global PSW key before updating the specified global area. If the tagname is synchronizable, the equivalent of a `SYNCC SYNC` assembler macro is performed to synchronize the global area across the loosely coupled complex; otherwise, if the tagname is keypointable, the equivalent of a `GLOUC` assembler macro is performed to keypoint the global record specified by tagname (or the record containing the global field). In all cases, this function returns with the PSW key set for working storage.

## Examples

The following example updates a non-synchronizable global record.

```
#include <tpfglbl.h>
#include <c$globz.h>
...
{
    /******
    /* Increment data element mydata in non-synchronizable global    */
    /* record _myglob.                                                */
    /******
    struct myglbrec **mgrptrptr = glob(_myglob);
    struct myglbrec *mgrptr = *mgrptrptr;
    long newdata = mgrptr->mydata + 1;
    glob_update(_myglob, &mgrptr->mydata, &newdata,
               sizeof mgrptr->mydata);
}
```

## Related Information

- “glob\_keypoint—Keypoint TPF Global Field or Record” on page 279
- “glob\_lock—Lock and Access Synchronizable TPF Global Field or Record” on page 280
- “glob\_modify—Modify TPF Global Field or Record” on page 282
- “glob\_sync—Synchronize TPF Global Field or Record” on page 284
- “glob\_unlock—Unlock TPF Global Field or Record” on page 286
- “glob—Address TPF Global Field or Record” on page 277.

## global—Operate on TPF Global Field

This function addresses (by indirection), updates, synchronizes, and keypoints global fields. It provides a high-level interface to TPF globals.

**Note:** TPF development uses the following convention for globals. The @ symbol, used in assembler to begin all global field and record names, is not permitted in any C identifier. Use an underscore symbol (\_) in its place. All remaining characters in the global tag name are converted to lowercase. For more information about globals, refer to Appendix F, “GNTAGH User's Guide” on page 1437.

### Format

```
#include <tpfglbl.h>
#include <c$globz.h>
void *global(unsigned int tagname, enum t_glbl action, void *pointer);
```

#### tagname

This argument is treated as an unsigned integer that uniquely identifies the global field to be operated on, which must be defined in header file c\$globz.h.

#### action

The action to be performed against **tagname**. Use one of these defined terms:

##### GLOBAL\_UPDATE

Update the specified global field with the value specified by the argument **pointer**, whose length is determined by the defined length implicit in the **tagname** definition. All functions requested to be performed against the field (keypointing, processor synchronization, and others) are validated based on the definition of the tagname. If the field can be synchronized, a **GLOBAL\_LOCK** must be issued before **GLOBAL\_UPDATE**. For this subfunction, argument **pointer** is required. The **GLOBAL\_UPDATE** action will implicitly do a **GLOBAL\_UNLOCK** action; you do not need to explicitly issue **GLOBAL\_UNLOCK**.

##### GLOBAL\_MODIFY

Alter the specified global field with the value specified by the argument **pointer**, whose length is determined by the defined length implicit in the **tagname** definition. No additional functions are performed against the field (keypointing, processor synchronization, and others). To perform additional functions, you must either use the **GLOBAL\_UPDATE** action described above, or code subsequent calls to `global` specifying the other actions described below. If the field can be synchronized, a **GLOBAL\_LOCK** must be issued before using **GLOBAL\_MODIFY**. For this subfunction, argument **pointer** is required.

##### GLOBAL\_KEYPOINT

Causes the keypointing of the specified global field. For this subfunction, argument **pointer** is not required.

##### GLOBAL\_LOCK

Reserves the exclusive use of the specified global field and forces refreshing of the core copy from the most current file copy. The contents of the specified global field are copied to user storage (pointed to by argument **pointer**) for the defined length of the global field. For this subfunction, argument **pointer** is required. You should ensure that a lock is held for the shortest possible time because severe system degradation may result from other ECBs waiting for locks. This action is valid only for synchronizable fields.



**GLOBAL\_UNLOCK**

Inverse of **GLOBAL\_LOCK**, which was explained previously. Releases exclusive use of the specified global field. For this subfunction, argument **pointer** is not required. You do not need to issue **GLOBAL\_UNLOCK** after a **GLOBAL\_UPDATE** action because **GLOBAL\_UPDATE** implicitly does a **GLOBAL\_UNLOCK**.

A **GLOBAL\_LOCK** must be issued before issuing **GLOBAL\_UNLOCK**. This action is valid only for synchronizable fields.

**GLOBAL\_COPY**

Copies the contents of the specified global field to user storage (pointed to by argument **pointer**) for the defined length of the global field. For this subfunction, argument **pointer** is required.

**GLOBAL\_SYNC**

Same as **GLOBAL\_UNLOCK** as described previously, except that synchronization of globals across processors in a loosely coupled complex is performed. For this subfunction, argument **pointer** is not required.

A **GLOBAL\_LOCK** must be issued before issuing **GLOBAL\_SYNC**

**pointer**

This argument is optional (unless you specify **GLOBAL\_UPDATE**, **GLOBAL\_MODIFY**, **GLOBAL\_LOCK**, or **GLOBAL\_COPY**), and is treated as a pointer to type void. Its context depends on the content of argument **action** as described previously, and may be considered either a source or destination of field content.

**Normal Return**

Void.

**Error Return**

Not applicable.

**Programming Considerations**

- The argument coded as **tagname** must be defined in header file `c$globz.h`. Results will be unpredictable if this restriction is not heeded.
- Only operations against global fields are supported; global records cannot be operated on. If the global tagname indicates that the specified global field is a record and not a field, a system error with exit results.
- Programs using the **GLOBAL\_LOCK** option should be prepared to issue a **GLOBAL\_UNLOCK** as soon as possible to prevent severe system degradation caused by the possibility of other ECBs awaiting locks on the target global block.
- Programs using the **GLOBAL\_LOCK**, **GLOBAL\_UNLOCK**, or **GLOBAL\_SYNC** options should not have any pending I/O operations outstanding because these functions may perform the equivalent of a `waitc` function.
- A program should not give up control (by calling an I/O function, such as `finwc`) between reading and updating a global field. If it does, the data that the program reads may have already been updated by another program by the time the program decides to update it. Use of the locking facility provides additional security when updating synchronizable global fields.

## Examples

```
#include <tpfglbl.h>
int nsIns;
...
global(_nsIns, GLOBAL_LOCK, &nsIns);    /* Lock and get copy of */
/* synchronizable global */
nsIns = nsIns+1                          /* Update value. */
global(_nsIns, GLOBAL_UPDATE, &nsIns);  /* Issue UPDATE.      */
...

```

## Related Information

292 TPF V4R1 C/C++ Language Support User's Guide

## gsysc—Get Storage from the System Heap

This function allocates storage from the system heap.

### Format

```
#include <sysapi.h>
void *gsysc(int frames, char *token);
```

#### frames

This argument specifies the number of 4-KB frames to be allocated from the system heap.

#### token

This argument specifies an 8-byte string that identifies the storage being allocated. You need to specify this string when releasing storage to verify that the correct storage is released.

### Normal Return

The system virtual memory (SVM) address of the storage requested is returned.

### Error Return

If the request cannot be satisfied because the maximum size of the heap has been reached or there is not enough contiguous space available, 0 is returned.

If the system does not have enough real frames to satisfy the request, a catastrophic dump is taken and the system is IPLed again.

## Programming Considerations

- Any storage allocated by `gsysc()` must be returned by the application using `rsysc()` because the storage is not attached to the ECB and will not be returned when the ECB exits.
- The storage key is set to 'XC'.

## Examples

The following example allocates and releases 12 KB of system heap storage.

```
#include <sysapi.h>
#include <stdio.h>
:
:
{
    /******
    /* Allocate 12 KB of storage for a table used by many different
    /* ECBs.
    /******
    int frames, rc;
    char * token = "TABLE40 ";
    struct table {
        char *name;
        int code;
    } *tbl_ptr;

    frames = 3;
    tbl_ptr = gsysc(frames, token);
    if (tbl_ptr == 0) {
        serrc_op(SERRC_EXIT, 0x1111, "Error allocating table.", NULL);
    }
:
rc = rsysc((void *)tbl_ptr, frames, token);
```

## gsysc

```
        if (rc == RSYSC_ERROR) {  
            serrc_op(SERRC_EXIT, 0x2222, "Error releasing storage.", NULL);  
        }  
    }
```

## Related Information

“rsysc—Release Storage from the System Heap” on page 443.

## inqrc—Convert Resource Application Interface

This function is used to convert communications resource identifiers and names. The function uses structures found in `tpfapi.h` for input and output parameter passing.

### Format

```
#include <tpfapi.h>
struct inqrc_return *inqrc(unsigned char req_type, union inqrc_arg *i_req);
```

**req\_type**  
Code **R** if the resource identifier (RID) or the session control block identifier (SCB ID) is to be used as a search argument, or **N** if the network qualified name (NQN) is to be used as a search argument.

**i\_req**  
Pointer to union of type `inqrc_arg`, which may contain an `r_type` structure containing the RID, an `r3_type` structure containing the 3-byte RID or SCB ID, or an `n_type` structure containing the NQN. This union is defined in `tpfapi.h`.

### Normal Return

Pointer to structure of type `inqrc_return`, defined in `tpfapi.h`.

### Error Return

Not applicable.

### Programming Considerations

- You must reserve 30 bytes for the input area (**i\_req**) because this area is used for the return structure.
- Specifying an invalid **type** parameter results in a system error with exit.
- If no network name is included on an NQN request, the first match on the name is returned regardless of the network name in the resource vector table (RVT).

### Examples

The following example requests the RVT data for the 3-byte RID at EBW041 and performs a check to ensure that the RID is valid.

```
#include <tpfapi.h>
#include <c$rc0pl.h>
struct rc0pl *rc0plptr;
union inqrc_arg *reqptr;
:
rc0plptr = (struct rc0pl *) &(ecbptr()->ebw000);
/* set up pointer to RCPL */
reqptr = (union inqrc_arg *) &(ecbptr()->ebw041);
/* storage for r_type request */
reqptr->r3_type.inqars2 = 0;
/* reserved */
reqptr->r3_type.inqarid3 = rc0plptr->dest.sna.rcpldes3;
/* set up RID */
(inqrc('R', reqptr)->inqrtncd
 ? serrc_op(SERRC_EXIT, 0x123456, "Invalid INQRC Request", NULL) \
 : serrc_op(SERRC_RETURN, 0x111111, "RID was Found", NULL));
```

### Related Information

“selec—Select a Thread Application Interface” on page 448.

---

## keyrc—Restore Protection Key

This function restores the current program status word (PSW) protection key to its normal value (protect key of working storage).

### Format

```
#include <tpfapi.h>
void      keyrc(void);
```

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- This function can be useful in restoring the normal storage protection key after a previous call to the `cinfo()` function in which the `CINFO_WRITE` option was specified.
- Your program should restore the normal storage protection key once it no longer needs to be able to write into areas in protected storage. If you do not restore the normal storage protection key, a subsequent error in your program could corrupt some portion of protected storage.
- Once you have restored the normal storage protection key, if your program attempts to write into protected storage, a protection exception occurs; the entry is exited and the 000003 system error is issued.

## Examples

This example restores the storage protection key after a call to the `cinfo` function in which the `CINFO_WRITE` option was specified.

```
#include <tpfapi.h>
.
.
.
char *field_ptr;
.
. /* can only read protected storage */
.
field_ptr = cinfo(CINFO_WRITE, CINFO_CMMINC);
.
. /* can read and write protected storage */
.
keyrc();
.
. /* can only read protected storage */
.
```

## Related Information

“`cinfo`—Control Program Interface” on page 38.

## keyrc\_okey—Restore Original Protection Key

This function restores the current program status word (PSW) protection key to its original value before the previous `cinfo` function in which the `CINFO_WRITE` option was specified.

### Format

```
#include <tpfapi.h>
void keyrc_okey(void);
```

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- When writing TPF 4.1 code to support application code that interfaces with system code or provides new C functions, it is necessary to preserve the application environment. One environmental consideration is the storage protection key. It may not be correct to assume that the caller is always processing with the working storage protection key. This function can be useful in restoring the protection key, which was in effect before the previous call to the `cinfo` function in which the `CINFO_WRITE` option was specified.
- Your C function should restore the original storage protection key once it no longer needs to write into areas in protected storage.
- Programs require restricted authorization to use this function; however, they do not need key 0 authorization.

## Examples

This example restores the original protection key of the calling function.

```
#include <tpfapi.h>
.
. /* called by an application to perform some service */
.
.
field_ptr = cinfo(CINFO_WRITE, CINFO_CMMINC);
.
. /* Can read and write protected storage to complete service */
. /* request, cinfo has saved the callers protection key in */
. /* CE20KEY. If any additional CINFO_WRITE request is issued */
. /* it will be necessary to save the first value in CE20KEY */
. /* and restore it prior to the keyrc_okey(). */
.
keyrc_okey();
.
. /* returned to callers protection key */
. /* */
```

## Related Information

“`cinfo`—Control Program Interface” on page 38.

## kill—Send a Signal to a Process

This function permits signals to be sent to another entry control block (ECB).

### Format

```
#define _POSIX_SOURCE
#include <signal.h>
pid_t kill(pid_t pid, int sig);
```

#### pid

The process identifier (ID) of the process that receives the signal from the `kill` function.

#### sig

One of the macros defined in the `signal.h` header file or 0. The signals that are supported are listed in Table 13 on page 495. If you specify a value of 0, error checking is performed, but a signal is not sent. Code **sig** as 0 to check whether the **pid** argument is valid.

### Normal Return

If successful, a value of 0 is returned.

### Error Return

If unsuccessful, the `kill` function returns a value of `-1` and sets `errno` to one of the following:

<b>EINVAL</b>	The value of the <b>sig</b> argument is an incorrect or unsupported signal number.
<b>EPERM</b>	The process does not have permission to send the signal to the receiving process.
<b>ESRCH</b>	The process specified by the <b>pid</b> parameter cannot be found.

### Programming Considerations

- For a process to have permission to send a signal to another process, the real or saved set-user-ID (UID) of the sending process must match the real or saved set-user-ID of the receiving process, or the sending process must have superuser privileges.
- Some TPF system events trigger the internal creation of a signal; for example, the exit of a child ECB that was created by the `tpf_fork` function results in a **sig** `SIGCHLD` returned to the parent.

### Examples

The following example shows how to send **sig** `SIGTERM`.

```
#include <stdio.h>
#include <stdlib.h>
#define _POSIX_SOURCE
#include <signal.h>
...
pid_t child_pid;
/* Create a child process*/
...
child_pid = tpf_fork(&create_parameters);
...
/* Signal child process to terminate*/
if (kill (child_pid, SIGTERM) == -1) {
```



```
if (errno == ESRCH) {  
    printf("Could not find child");  
    abort();  
}  
}
```

## Related Information

- “alarm—Schedule an Alarm” on page 13
- “sigaction—Examine and Change Signal Action” on page 486
- “signal—Install Signal Handler” on page 495
- “sigpending—Examine Pending Signals” on page 498
- “sigprocmask—Examine and Change Blocked Signals” on page 499
- “sigsuspend—Set Signal Mask and Wait for a Signal” on page 502
- “tpf\_fork—Create a Child Process” on page 594
- “tpf\_process\_signals—Process Outstanding Signals” on page 622
- “wait—Wait for Status Information from a Child Process” on page 684
- “waitpid—Obtain Status Information from a Child Process” on page 687.

## levtest—Test Core Level for Occupied Condition

This function tests the indicated core block reference word (CBRW) data level or data event control block (DECB) for the existence of a working storage block.

### Format

```
#include <tpfapi.h>
int levtest(enum t_lvl level);
```

or

```
#include <tpfapi.h>
int levtest(TPF_DECB *decb);
```

#### level

One of 16 possible values representing a valid entry control block (ECB) data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter identifies the CBRW to be tested for the presence of a working storage block.

#### decb

A pointer to a DECB. This parameter identifies the DECB to be tested for the presence of a working storage block.

### Normal Return

An integer value representing the size of the working storage block in bytes if the ECB data level or DECB is occupied, or zero if the ECB data level or DECB is unoccupied.

### Error Return

Not applicable.

## Programming Considerations

- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- If the DECB pointer does not point to a valid DECB, the source routine will take a SERRC (which is a standard memory dump of main storage) and exit the ECB.

## Examples

The following example obtains and addresses a working storage block on ECB data level DC if it is unoccupied, or simply addresses the existing block if occupied.

```
#include <tpfapi.h>
char *work;
:
:
if(!levtest(DC))
    work = getcc(DC, TYPE, L2);
else
    work = ecbptr()->ce1crc;
```

The following example obtains the address of a working storage block on an unoccupied DECB.

```
#include <tpfapi.h>
char *work;
TPF_DECB *decb
:
:
if(!levtest(decb))
    work = (char *)getcc(decb, GETCC_TYPE, L2);
```

## Related Information

- “crusa—Free Core Storage Block If Held” on page 73
- “getcc—Obtain Working Storage Block” on page 248
- “relcc—Release Working Storage Block” on page 421.

See *TPF Application Programming* for more information about DECBs.

## link—Create a Link to a File

This function provides an alternative path name for an existing file and afterwards, the same file can be accessed using both path names. The `link` function creates a new link from the new path name to the file and the link may be in the same or a different directory.

### Format

```
#include <unistd.h>
int link(const char *oldfile, const char *newname);
```

#### **oldfile**

The path name of the existing file.

#### **newname**

The path name to be created for the new link to the existing file.

This function provides an alternative path name for the existing file so that the file can be accessed by either the old or the new name. The `link` function creates a link from path name **newname** to an existing file with path name **oldfile**. The link can be stored in the same directory as the original file or in a completely different one.

Links are allowed to files only, not to directories.

This is a hard link, which ensures the existence of a file even after its original name has been removed.

If the `link` function successfully creates the link, it increments the link count of the file. The link count tells how many links there are to the file.

#### **TPF deviation from POSIX**

The TPF system does not support the `atime` (access time) or `ctime` (change time) time stamp.

If **oldfile** names a symbolic link, the `link` function creates a link that refers to the file that results from resolving the path name contained in the symbolic link. If **newname** names a symbolic link, the `link` function fails and sets `errno` to `EEXIST`.

### Normal Return

If successful, the `link` function returns a value of 0.

### Error Return

If unsuccessful, the `link` function returns a value of `-1` and sets `errno` to one of the following:

<b>EACCES</b>	The process does not have appropriate permissions to create the link. Possible reasons include no search permission on a path name component of <b>oldfile</b> or <b>newname</b> , no write permission on the directory intended to contain the link.
<b>EEXIST</b>	Either <b>newname</b> refers to a symbolic link, or a file or directory with the name <b>newname</b> already exists.
<b>EINVAL</b>	Either <b>oldfile</b> or <b>newname</b> is incorrect.

<b>ELOOP</b>	A loop exists in symbolic links. This error is issued if the number of symbolic links found while resolving the <b>oldfile</b> or <b>newname</b> argument is greater than the POSIX_SYMLINK_MAX value.
<b>ENAMETOOLONG</b>	<b>oldfile</b> or <b>newname</b> is longer than the PATH_MAX value, or a component of one of the path names is longer than the NAME_MAX value. For symbolic links, the length of the path name string substituted for a symbolic link in <b>oldfile</b> or <b>newname</b> exceeds the PATH_MAX value.
<b>ENOENT</b>	A path name component of <b>oldfile</b> or <b>newname</b> does not exist, <b>oldfile</b> itself does not exist, or one of the two arguments is an empty string.
<b>ENOSPC</b>	The directory intended to contain the link cannot be extended to contain another entry.
<b>ENOTDIR</b>	A path name component of one of the arguments is not a directory.
<b>EPERM</b>	<b>oldfile</b> is the name of a directory and links to directories are not supported.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

Because the link function requires an update to a directory file, new links cannot be created in 1052 or UTIL state.

## Examples

The following example provides link information for a file.

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

main() {
    char fn[]="link.example.file";
    char ln[]="link.example.link";
    int fd;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        if (link(fn, ln) != 0) {
            perror("link() error");
            unlink(fn);
        }
        else {
            unlink(fn);
            unlink(ln);
        }
    }
}
```

**link**

## **Related Information**

- “rename—Rename a File” on page 428
- “symlink—Create a Symbolic Link to a Path Name” on page 521
- “unlink—Remove a Directory Entry” on page 668.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## lockc—Lock a Resource

This function is used to lock a resource from access by another I-stream. If the lock is held by this I-stream, a dump is taken and the lock updated to show this occurrence. If the lock is held by another I-stream, the code will spin for a second or return a condition code depending on the `t_lock_opt` value. If the spin ends successfully, a catastrophic dump is taken. If the lock is free, this code will lock it.

### Format

```
#include <tpfapi.h>
int lockc(void *lockword, enum t_lock_opt opt);
```

#### lockword

A pointer to a *lock* made up of 2 consecutive fullwords used for lock and trace.

#### opt

One of 2 possible values directing the lockc function to either spin or return a condition code when the lock is held by another I-stream. The possible values of the enumeration type `t_lock_opt` are `LOCK_O_SPIN` and `LOCK_O_RETN`.

### Normal Return

Zero.

### Error Return

4 Lock is being held by another I-stream (for `LOCK_O_RETN` only)

## Programming Considerations

- The lock specified by `lockword` must not be held by this I-stream. If the lock is held, a 000572 system error will be issued and the lock held (for `LOCK_O_SPIN` only).
- Any other lock should not be held by this I-stream because a "lock-out" condition could occur. This implies that control must not be given up once a lock is held.
- A catastrophic dump will occur if the spin lock (`LOCK_O_SPIN`) times out (the 000571 system error).

## Examples

The following example attempts to lock an area in which the lock field is pointed to by `table_lock_ptr`. The return condition is checked.

```
#include <tpfapi.h>
...
if (lockc(table_lock_ptr, LOCK_O_RETN) != 0)
...

```

## Related Information

"`unlk`—Unlock a Resource" on page 670.

## lodic–Check System Load and Mark ECB

This function is used to check if enough system resources are available to begin processing low-priority or batch work, and to determine if an entry control block (ECB) can be suspended (based on the level of available resources).

An ECB that calls the `lodic` function is marked as a low-priority ECB. Once marked, the ECB can be suspended when system resources are below the shutdown levels defined for the BATCH priority class (see Table 10 on page 310). Even though the ECB is marked as being able to be suspended, the ECB cannot be suspended until it gives up control. Once the ECB has been suspended, it will not receive control again until enough system resources are available.

### Format

```
#include <tpfapi.h>
int lodic();
```

### Normal Return

The return code depends on the shutdown levels that are defined for the BATCH priority class (see Table 10 on page 310).

- 0 The available system resources are below the shutdown levels defined for the BATCH priority class (more work will not be started).
- 1 The available system resources are above the shutdown levels defined for the BATCH priority class (more work is allowed to be started).

### Error Return

Not applicable.

## Programming Considerations

- This function can run on any I-stream.
- Once an ECB has been marked as capable of being suspended, it becomes suspended whenever the ECB gives up control and the available system resources are below the defined shutdown levels (unless the ECB is holding a resource). Once suspended, the ECB will remain suspended until the available system resources return to levels above the defined shutdown levels.
- The create functions, `cremc`, `credc`, `swisc_create`, `crexc`, and `creec`, will pass the `lodic` parameters from the parent ECB to the child ECB. If the parent ECB is marked as being able to be suspended, the child ECB is also marked in the same way. The other create functions, `cretc` and `cretc_level`, do not pass this information.
- When you use a function to create an ECB (such as `cremc` or `swisc_create`), the ECB is not created immediately. First, a system work block (SWB) is placed on a list in the CPU loop. The ECB is not actually created until the SWB is dispatched from the list. Because of this delay, call the `lodic` function each time before creating a single ECB. The application calling the `lodic` function should give up control before calling the `lodic` function again to ensure that ECBs that are created are given a chance to receive control and perform work. Otherwise, system resource depletion can result from too much work being scheduled and not started.
- Suspended ECBs that are using a large amount of resources can result in conditions in which these ECBs never get back control. Avoid having an ECB become suspended if it is using a large amount of resources.



- Use the `lodlc_ext` function to prevent this ECB from becoming suspended after you run the `lodlc` function.
- ECBs that are suspended because of a `lodlc` call will be purged during cycle-down to 1052 state unless they have been previously identified to survive cycle-down.

## Examples

The following example creates a new ECB after checking to see if available resources exist. The ECB is also marked as a low-priority batch ECB and is allowed to be suspended while resources are not held.

```
#include <tpfapi.h>
void QZZ0();
:
rc = lodlc();
if (rc) credc( 0, NULL, QZZ1 );
```

## Related Information

"`lodlc_ext`—Check System Load and Mark ECB with Extended Options" on page 308.

## lodict\_ext–Check System Load and Mark ECB with Extended Options

The lodict\_ext function is used to check if enough system resources are available to begin processing low-priority or batch work, and to determine if an ECB can be suspended (based on the level of available resources).

An ECB that calls the lodict\_ext function with the **LODIC\_ECBCREATE** or **LODIC\_SUSPEND** parameters is marked as a low-priority ECB. Once marked, the ECB can be suspended when system resources are below the shutdown levels defined for a specified priority class (see Table 10 on page 310). Even though the ECB is marked as being able to be suspended, the ECB cannot be suspended until it gives up control. In most cases the **LODIC\_SUSPEND** parameter forces the ECB to immediately give up control. The **LODIC\_ECBCREATE** parameter does not force the ECB to immediately give up control; the ECB must wait until it gives up control by entering a post-interrupt routine. Once suspended, the ECB does not receive control again until enough system resources are available.

### Format

```
#include <tpfapi.h>
int lodict_ext(int flags, const void *usrprm);
```

#### flags

The following flags are defined in tpfapi.h:

#### **LODIC\_ECBCREATE, LODIC\_SUSPEND, LODIC\_UNMARK, or LODIC\_CHECK**

Choose one of the following four parameters.

##### **LODIC\_ECBCREATE**

The ECB is marked as suspendable and a priority class is assigned to the ECB.

There is no immediate loss of control if you use the **LODIC\_ECBCREATE** parameter.

Checks are performed to see if the available system resources are above the shutdown levels defined by the specified priority class (for example, checks to see if more work can be started).

**0** Indicates that system resources are low and that more work cannot be started.

**1** Indicates that more work can be started.

##### **LODIC\_SUSPEND**

The ECB is marked as capable of being suspended, and a priority class is assigned to the ECB.

The **LODIC\_SUSPEND** parameter will cause the ECB to immediately lose control unless the ECB is holding a resource and **LODIC\_HOLD** is not coded.

Checks are performed to see if the available system resources are above the shutdown levels defined by the specified priority class (for example, checks to see if more work can be started). If the available system resources are below the shutdown levels and the ECB is able to lose control, the ECB is not immediately suspended. In all cases but one, a return code of 1 will be returned. A return code of 0 is returned in the case where the ECB does not lose control and the system resources available are below the shutdown levels.

**0** More work cannot be started.

**1** More work can be started.

### **LODIC\_UNMARK**

Removes the ability for the ECB to become suspended.

**Note:** An ECB marked as being capable of being suspended remains marked as such until you enter **LODIC\_UNMARK** or until the ECB exits.

### **LODIC\_CHECK**

Checks are performed to see if the available system resources are above the shutdown levels defined by the specified priority class (for example, more work can be started). There is no immediate loss of control if the **LODIC\_CHECK** parameter is used. The **LODIC\_CHECK** parameter does not mark or unmark the ECB as an ECB that can be suspended.

### **LODIC\_BATCH, LODIC\_LOBATCH, LODIC\_IBMHI, or LODIC\_IBMLO**

Choose only one of the four priority classes.

A priority class is assigned to the ECB.

- **LODIC\_BATCH** is reserved for users.
- **LODIC\_LOBATCH** is reserved for users.
- **LODIC\_IBMHI** is reserved for IBM written E-type programs.
- **LODIC\_IBMLO** is reserved for IBM written E-type programs.

A priority class is used with the **LODIC\_ECBCREATE**, **LODIC\_SUSPEND**, and **LODIC\_CHECK** parameters.

**Note:** See Table 10 on page 310 for the shutdown levels of each priority class.

### **LODIC\_HOLD**

Indicates if the ECB can be suspended while holding a resource (fiwhc, corhc, tasnc, evnwc, or glob\_lock). **LODIC\_HOLD** can be used with **LODIC\_ECBCREATE** and **LODIC\_SUSPEND**.

#### **Notes:**

1. This parameter does not apply to resources held with the lockc function.
2. Coding **LODIC\_HOLD** requires restricted authorization (OPTIONS=(RESTRICT)) in the IBMPAL macro.

### **usrprm**

Points to user data that will be passed to the LODIC user exit. Otherwise, coding NULL is acceptable.

## **Normal Return**

The return code depends on the shutdown levels that are defined for the specified priority class (see Table 10 on page 310).

- 0** Indicates that the available system resources are below the shutdown levels defined for the specified priority class (more work should not be started).
- 1** Indicates that the available system resources are above the shutdown levels defined for the specified priority class (more work can be started).

## lodict\_ext

These return codes do not apply to the **LODIC\_UNMARK** parameter.

## Error Return

Not applicable.

## Programming Considerations

- Once an ECB has been marked as capable of being suspended, it becomes suspended whenever the ECB gives up control and the available system resources are below the defined shutdown levels (unless the ECB is holding a resource and **LODIC\_HOLD** is not coded). Once suspended, the ECB remains suspended until the available system resources return to levels above the defined shutdown levels.
- The create functions `cremc`, `credc`, `swisc_create`, `crexc`, and `creec`, will pass the `lodict_ext` options and the `lodict_ext` priority class information from the parent ECB to the child ECB. If the parent ECB is marked as capable of being suspended, the child ECB is also marked in the same way. The other create functions, `cretc` and `cretc_level`, do not pass this information.
- When you use a function to create an ECB (such as `cremc` or `swisc_create`), the ECB is not created immediately. First, a system work block (SWB) is placed on a list in the CPU loop. The ECB is not actually created until the SWB is dispatched from the list. Because of this delay, call the `lodict_ext` function each time before you create a single ECB. The application calling `lodict_ext` must give up control before calling `lodict_ext` again to help ensure that ECBs that are created are given a chance to receive control and perform work. Otherwise, system resource depletion can result from too much work scheduled but not started.
- ECBs that are marked with **LODIC\_HOLD** may hold resources while suspended.
- Suspended ECBs that are using a large amount of resources can result in conditions in which these ECBs never get back control. Avoid having an ECB become suspended if it is using a large amount of resources.
- ECBs that are suspended because of a `lodict` call will be purged during cycle-down to 1052 state unless they have been previously identified to survive cycle-down.
- Table 10 provides the shutdown levels of each priority class. The values given are the minimum percentage of each block type that must be available for ECBs to continue running.

In general, when the availability of a particular block type falls below a shutdown percentage defined in Table 10, ECBs labeled with the associated priority class will be suspended and will remain suspended until the availability of that block type rises above the shutdown percentage.

Table 10. Priority Class Table (Shutdown Levels)

Priority Class	Block Type				
	CMB	ECB	FRM	IOB	SWB
LOBATCH	96%	96%	96%	96%	96%
BATCH	48%	48%	48%	48%	48%
IBMLO	96%	96%	96%	96%	96%
IBMHI	32%	32%	32%	32%	32%

- These are the shutdown level values as shipped by IBM.
- IBMLO and IBMHI are reserved for use by IBM.

## Examples

The following example creates a new ECB after checking to see if available resources exist. The ECB is also marked as a low-priority or batch ECB and is allowed to be suspended while holding resources. The address of a save area, called *passarea*, is passed to the LODIC user exit.

```
#include <tpfapi.h>
void QZZ0();
:
rc = lodict_ext( LODIC_ECBCREATE+LODIC_BATCH+LODIC_HOLD,
                &passarea );
if (rc) credc( 0, NULL, QZZ1 );
```

## Related Information

"lodict-Check System Load and Mark ECB" on page 306.

---

### longc–Set Entry Maximum Existence Time

This function allows the ECB-controlled program to set or reset the maximum allowed lifetime of the current entry in the system.

#### Format

```
#include <tpfapi.h>
void longc(unsigned int longc_req);
```

##### **longc\_req**

One of the following, specifying the lifetime for the ECB:

##### **LONGC\_INDEF**

requests that the ECB be allowed to exist indefinitely.

##### **LONGC\_CLEAR**

requests that the long life entry detection fields be reset. The entry is given a maximum life of one additional minute.

**n** where n is an integer in the range of 1–254, indicating the desired maximum lifetime of the entry in minutes.

#### Normal Return

Void.

#### Error Return

Not applicable.

### Programming Considerations

- Specifying an invalid parameter results in a system error with exit.
- The long life entry detection program is inhibited for all programs calling `longc(LONGC_INDEF)`.
- Specifying integer value n causes the long life entry detection program to flag the entry as looping if it is still active approximately (n + 1) minutes later.
- `longc(LONGC_INDEF)` can be used by an application program to inhibit long life entry detection for a specific transaction even though it was enabled by a previous program.
- The long life detection program sends a message to the system operator whenever an entry is found to have exceeded its maximum life in the system.

### Examples

The following example requests long life ECB detection to be inhibited for the current entry.

```
#include <tpfapi.h>
:
:
longc(LONGC_INDEF);          /* Inhibit long life entry detection */
```

### Related Information

None.

## longjmp–Restore Stack Environment

This function restores a stack environment previously saved in **env** by the `setjmp` function. The `setjmp` and `longjmp` functions provide a way to perform a nonlocal goto. They are often used in signal handlers.

A call to the `setjmp` function causes the current stack environment to be saved in **env**. A subsequent call to the `longjmp` function restores the saved environment and returns control to a point in the program corresponding to the `setjmp` call. The program resumes as if the `setjmp` call had just returned the given value of the **value** argument. The `setjmp` and `longjmp` functions are not restricted to the same dynamic load module (DLM).

### Format

```
#include <setjmp.h>
void longjmp(jmp_buf env, int value);
```

#### **env**

The current stack environment previously saved by the `setjmp` function.

#### **value**

The value of the variable when the `longjmp` function was called. **value** must be nonzero; otherwise, the `longjmp` function substitutes a 1 in its place.

### Normal Return

Void.

### Error Return

Error conditions exist that would cause the process to exit with a SNAPC dump; the ones that can be detected by the `longjmp` function are:

- The stack integrity check failed because the caller's entry point saved in the jump buffer at `setjmp` time did not match the caller's entry point at `longjmp` time.
- The target Program Nesting Level (PNL) to which the `longjmp` function should return does not exist in the current PNL.

## Programming Considerations

- Variables that are accessible to the function that receives control contain the values they had when the `longjmp` function was called.
- The values of register variables are unpredictable. Nonvolatile **auto** variables that are changed between calls to the `setjmp` and `longjmp` functions are also unpredictable.
- The database ID (DBI), program base ID (PBI), subsystem user (SSU) ID, or any other application program global pointer that might have changed after the `setjmp` function is issued needs to be restored by users prior to calling the `longjmp` function.
- Ensure that the function that calls the `setjmp` function does not return before you call the corresponding `longjmp` function. Calling the `longjmp` function after the function calling `setjmp` returns causes unpredictable program behavior.

### Examples

This example provides for saving the stack environment at the following statement:  
`if(setjmp(mark) != 0) ...`

## longjmp

When the example first performs the `if` statement, it saves the environment in *mark* and sets the condition to `FALSE` because the `setjmp` function returns a 0 when it saves the environment. The program prints the message: `setjmp has been called`.

The subsequent call to function *p* tests for a local error condition, which can cause it to perform the `longjmp` function. (The function *p* that performs the `longjmp` function can be in the same DLM or a separate DLM). Control then returns to the original `setjmp` function using the environment saved in *mark*. This time the condition is `TRUE` because -1 is the returned value from the `longjmp` function. The example then performs the statements in the block and prints: `longjmp has been called`.

It then performs the `recover` function and leaves the program.

```
/* Illustration of longjmp(). */
#include <stdio.h>
#include <setjmp.h>
jmp_buf mark;
void p(void);
void recover(void);
int ADLM(void)
{
    if (setjmp(mark) != 0)
    {
        printf("longjmp has been called\n");
        recover();
        return(1);
    }
    printf("setjmp has been called\n");
    :
    :
    p();
    :
    :
    return(32);    /* Return suitable value */
}
void p(void)
{
    int error = 0;
    :
    :
    error = 9;
    :
    :
    if (error != 0)
        longjmp(mark, -1);
    :
    :
}
void recover(void)
{
    :
    :
}
```

## Related Information

“`setjmp`—Preserve Stack Environment” on page 470.



## lseek—Change the Offset of a File

This function sets the file offset.

### Format

```
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int pos);
```

#### **fildes**

The file descriptor whose file offset you want to set.

#### **off\_t offset**

The amount (positive or negative) that the byte offset will be changed. The sign (+ or –) indicates whether the offset is to be moved forward (positive) or backward (negative).

#### **pos**

The position in the file to start the change.

Specify one of the following symbols (defined in the `unistd.h` header file):

#### **SEEK\_SET**

The start of the file.

#### **SEEK\_CUR**

The current file offset in the file.

#### **SEEK\_END**

The end of the file.

The new position is the byte **offset** from the position specified by **pos**. After you have used the `lseek` function to seek to a new location, the next I/O operation on the file begins at that location.

The `lseek` function lets you specify new file offsets beyond the current end of the file. If data is written at such a point, read operations in the gap between this data and the old end of the file will return bytes containing zeros. (In other words, the gap is assumed to be filled with zeros.)

Seeking past the end of a file, however, does not automatically extend the length of the file. There must be a write operation before the file is actually extended.

### Normal Return

If successful, the `lseek` function returns the new file offset, measured in bytes, from the beginning of the file.

### Error Return

If unsuccessful, the `lseek` function returns a value of –1 and sets `errno` to one of the following:

#### **EBADF**

**fildes** is not a valid open file descriptor.

#### **EINVAL**

**pos** contained something other than one of the three options, or the combination of the **pos** and **off\_t offset** values would have placed the file offset before the beginning of the file, or the size of the file cannot be determined.

## lseek

### Programming Considerations

- Offsets are associated with one or more file descriptors and the corresponding open file description rather than with the file itself; the same file may have alternative offsets depending upon which file descriptor is used to access the file.
- If the size of character special files cannot be determined, the `lseek` function fails with `errno` set to `EINVAL`.

### Examples

The following example positions a file (that has at least 10 bytes) to an offset of 10 bytes before the end of the file.

```
lseek(fildes,-10,SEEK_END);
```

### Related Information

- “`fseek`—Change File Position” on page 226
- “`fsetpos`—Set File Position” on page 228
- “`creat`—Create a New File or Rewrite an Existing File” on page 54
- “`dup`—Duplicate an Open File Descriptor” on page 97
- “`fcntl`—Control Open File Descriptors” on page 129
- “`open`—Open a File” on page 380
- “`read`—Read from a File” on page 410
- “`write`—Write Data to a File Descriptor” on page 696.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## lstat—Get Status of a File or Symbolic Link

This function gets status information about a specified file.

### Format

```
#include <sys/stat.h>
int lstat(const char *pathname, struct stat *buf);
```

#### pathname

The path name of the file or symbolic link.

#### buf

The address of the struct stat object where information is to be stored.

After the `lstat` function gets status information about a specified file, the function places this information in the area of storage pointed to by the **buf** argument. You do not need permissions on the file itself, but you must have search permission on all directory components of **pathname**.

If the named file is a symbolic link, the `lstat` function returns information about the symbolic link itself.

See Table 15 on page 513 for more information about the `stat` structure, which is defined in the `sys/stat.h` header file.

#### TPF deviation from POSIX

The TPF system does not support the `atime` (access time) or `ctime` (change time) time stamp.

You can examine properties of a `mode_t` value from the `st_mode` field by using a collection of macros defined in the `modes.h` header file. If `mode` is a `mode_t` value from the `stat` structure:

**S\_ISBLK(mode)** Is nonzero for block special files.

#### TPF deviation from POSIX

The TPF system does not support block special files; this macro is included for compatibility only.

**S\_ISCHR(mode)** Is nonzero for character special files.

**S\_ISDIR(mode)** Is nonzero for directories.

**S\_ISFIFO(mode)** Is nonzero for pipes and first-in-first-out (FIFO) special files.

**S\_ISLNK(mode)** Is nonzero for symbolic links.

**S\_ISREG(mode)** Is nonzero for regular files.

**S\_ISSOCK(mode)** Is nonzero for sockets.

**TPF deviation from POSIX**

TPF sockets are non-standard, and are not associated with links or standard file descriptors; therefore, it is not possible to get status for a TPF socket. This macro is included for compatability only.

If the `lstat` function successfully determines all this information, it stores the information in the area indicated by the **buf** argument.

**Normal Return**

If successful, the `lstat` function returns a value of 0.

**Error Return**

If unsuccessful, the `lstat` function returns a value of -1 and sets `errno` to one of the following:

<b>EACCES</b>	The process does not have search permission on some component of the <b>pathname</b> prefix.
<b>EINVAL</b>	<b>buf</b> contains a null pointer.
<b>ELOOP</b>	A loop exists in symbolic links. This error is issued if the number of symbolic links found while resolving the <b>pathname</b> argument is greater than the <code>POSIX_SYMLINK_MAX</code> .
<b>ENAMETOOLONG</b>	<b>pathname</b> is longer than <code>PATH_MAX</code> characters or some component of <b>pathname</b> is longer than <code>NAME_MAX</code> characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds <code>PATH_MAX</code> .
<b>ENOENT</b>	There is no file named <b>pathname</b> , or <b>pathname</b> is an empty string.
<b>ENOTDIR</b>	A component of the <b>pathname</b> prefix is not a directory.

**Programming Considerations**

None.

**Examples**

The following example provides status information for a file.

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

main() {
    char fn[]="temp.file", ln[]="temp.link";
    struct stat info;
    int fd;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        if (link(fn, ln) != 0)
            perror("link() error");
    }
}
```

```

else {
    if (lstat(ln, &info) != 0)
        perror("lstat() error");
    else {
        puts("lstat() returned:");
        printf(" inode:  %d\n", (int) info.st_ino);
        printf(" mode:   %08x\n", info.st_mode);
        printf(" links:  %d\n", info.st_nlink);
        printf(" uid:    %d\n", (int) info.st_uid);
        printf(" gid:    %d\n", (int) info.st_gid);
    }
    unlink(ln);
}
unlink(fn);
}
}

```

### Output

```

lstat() returned:
inode:  3022
mode:   03000080
links:  2
uid:    255
gid:    256

```

## Related Information

- “chmod—Change the Mode of a File or Directory” on page 32
- “chown—Change the Owner or Group of a File or Directory” on page 35
- “creat—Create a New File or Rewrite an Existing File” on page 54
- “dup—Duplicate an Open File Descriptor” on page 97
- “fcntl—Control Open File Descriptors” on page 129
- “fstat—Get Status Information about a File” on page 230
- “link—Create a Link to a File” on page 302
- “mkdir—Make a Directory” on page 326
- “open—Open a File” on page 380
- “read—Read from a File” on page 410
- “readlink—Read the Value of a Symbolic Link” on page 417
- “remove—Delete a File” on page 427
- “symlink—Create a Symbolic Link to a Path Name” on page 521
- “unlink—Remove a Directory Entry” on page 668
- “write—Write Data to a File Descriptor” on page 696.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## mail–Process Internet Mail

This function allows you to process Internet mail, also known as electronic mail (e-mail), on the TPF Internet mail server. See *TPF Transmission Control Protocol/Internet Protocol* for more information about TPF Internet mail server support.

### Format

```
#include <tpf_mail.h>
void mail(char flag,
           char **rcpt,
           char *path,
           char *dmn,
           char *uid,
           char *pw,
           int rc);
```

#### flag

The function that you want to perform. Specify one of the following values:

- 'e' Checks if there is available mail to read.
- 'r' Reads Internet mail from the TPF Internet mail server and stores it in the file on the TPF file system specified by **path**.
- 'g' Reads Internet mail from the TPF Internet mail server and stores it in the TPF database. The file address of where the mail is stored is returned in the **rc** parameter.
- 'w' Sends Internet mail from the file specified by **path** to the TPF Internet mail server.
- 'p' Sends Internet mail from a file address specified by **path** to the TPF Internet mail server.

#### rcpt

A pointer to an array of pointers. Each pointer in the array points to a character string that contains the Internet mail address of the recipient. In the array, specify a NULL pointer to indicate the end of the recipient list.

This parameter applies only when the value for **flag** is 'w' or 'p'. Specify NULL for this parameter when the value for **flag** is 'r', 'g', or 'e'.

#### path

A pointer to a character string that contains the path name of a file or a file address. The value for **path** depends on the value you specify for **flag**, as follows:

- When you specify the value 'r' for **flag**, specify the path name for the file in which you want received (or read) Internet mail stored.
- When you specify the value 'g' for **flag**, specify NULL for this parameter.
- When you specify the value 'w' for **flag**, specify the path name for the file that contains the mail that you want to send.
- When you specify the value 'p' for **flag**, specify the file address for the mail that you want to send.
- When you specify the value 'e' for **flag**, specify NULL for this parameter.

#### dmn

A pointer to a character string that contains the name of the mail domain in which you have an Internet mail account.

**uid**

A pointer to a character string that contains your Internet mail account name.

**pw**

A pointer to a character string that contains the password for your Internet mail account.

**rc** The return code. See “Normal Return” and “Error Return” for more information.

## Normal Return

**rc** is one of the following values:

**0** One of the following:

- If the value specified for **flag** was '**e**', Internet mail is available for reading.
- If the value specified for **flag** was '**r**', '**w**', or '**p**', the function was completed successfully.

**1** One of the following:

- If the value specified for **flag** was '**e**', no Internet mail is available for reading.
- If the value specified for **flag** was '**r**' or '**g**', no Internet mail was found.

**fileaddr**

The file address of where the mail is stored. This is returned if the value specified for **flag** was '**g**' and the function was completed successfully.

## Error Return

The value of **rc** is **-1** and **errno** is set to the following:

**MAIL\_ERRNO**

The TPF Internet mail server detected a protocol error.

## Programming Considerations

- A single dot on a line (`\n.\n`) in your mail message specifies the end of the message. If you want to include a line in your message that contains just a single dot, code 2 dots as follows:

```
\n.. \n
```

- See the IMAIL data macro for information about the data format for the mail content.

## Examples

An Internet mail message is processed in the following example. In this example:

- An Internet mail message is built in a temporary file named `/temp/file`.
- An Internet mail message is sent to the users at addresses `moe@tpf.com` and `larry@tpf2.com`.
- The TPF Internet mail server is checked for incoming mail and, if found, stores the mail in a file named `/temp/mailin`.

```
/*
   Build an Internet mail message &
   Send an Internet mail message
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <tpf_mail.h>
#include <string.h>
```

## mail

```
#define ERROR -1
#define OK 0

int build_message(char *mail_message);

void main()
{
    int rc=0;
    char *rcpt[3];
    char mail_message[] = "/temp/file";
    char readmail_message[] = "/temp/mailin";
    char dmn[] = "tpf.com";
    char uid[] = "curly";
    char pw[] = "shemp3";

    rcpt[0]="moe@tpf.com";
    rcpt[1]="larry@tpf2.com";
    rcpt[2]=(char*)NULL;

    if((rc=build_message(mail_message)) == OK)
    {
        mail('w',rcpt,mail_message,dmn,uid,(char*)NULL,rc);
        if(rc==0)printf("Message sent\n");
        else{
            printf("Message not sent\n");
        }
    }
    else{
        printf("ERROR OCCURRED: CHECK ERRNO");
    }

    mail('e',(char*)NULL,(char*)NULL,dmn,uid,pw,rc);
    if (rc==0)
    {
        mail('r',(char*)NULL,readmail_message,uid,pw,rc);
    }
}

int build_message(char *mail_message)
{
    FILE* in;
    char content[]="Dear Moe\n\nHow's it going.\n\nRegards Curly.\n";
    int ret=OK;

    if((in=fopen(mail_message,"w"))==(FILE*)NULL)
    {
        fprintf(stderr,"Can't open message file: %s.\n",mail_message);
        return(ERROR);
    }
    fprintf(in,"%s",content);
    return OK;
}
```

The following example shows how you can process an Internet mail message using file addresses on the TPF database. In this example:

- An Internet mail message is built in a file record on the TPF database.
- An Internet mail message is sent to the users at addresses `moe@tpf.com` and `larry@tpf2.com`.
- The TPF Internet mail server is checked for incoming mail and, if found, stores the mail in a file on the TPF database. The file address of where the mail is stored is returned in `rc`.



```

/*
    Build an Internet mail message &
    Send an Internet mail message
*/

#include <stdio.h>
#include <stdlib.h>
#include <tpf_mail.h>
#include <string.h>

#define ERROR -1
#define OK 0

int build_message(char *);

void main()
{
    int rc=0;
    char *rcpt[3];
    char mail_message[16];
    char dmn[] = "tpf.com";
    char uid[] = "curly";
    char pw[] = "shemp3";

    rcpt[0]="moe@tpf.com";
    rcpt[1]="larry@tpf2.com";
    rcpt[2]=(char*)NULL;

    mail_message[0]='\0';

    if((rc=build_message(mail_message)) == OK)
    {
        mail('p',rcpt,mail_message,dmn,uid,(char*)NULL,rc);
        if(rc==0)printf("Message sent\n");
        else{
            printf("Message not sent\n");
        }
    }
    else{
        printf("ERROR OCCURRED: CHECK ERRNO");
    }

    mail('e',(char*)NULL,(char*)NULL,dmn,uid,pw,rc);
    if (rc==0)
    {
        mail('g',(char*)NULL,(char*)NULL,uid,pw,rc);
    }
}

int build_message(char *msg_FA)
{
    unsigned int fa;
    message_record *msgrec;

    fa = getfc(D8, GETFC_TYPE0, "\xFC\x60",
               (int)(GETFC_BLOCK+GETFC_FILL),GETFC_NOSERRC,0x00);

    sprintf(msg_FA,"%d",fa);

    ecbptr()->ebcid8 = 0xFC60;
    msgrec = ( message_record *) ecbptr()->ebccr8;

    if (msgrec == NULL)
        return (ERROR);
}

```

## mail

```
msgrec->MR0RID = 0XFC60;  
strcpy(&msgrec->msg_data,"Hi! How are you today?\n");  
return OK;  
}
```

## Related Information

- “close—Close a File” on page 44
- IMAIL data macro
- “open—Open a File” on page 380
- “read—Read from a File” on page 410
- “stat—Get File Information” on page 513
- “write—Write Data to a File Descriptor” on page 696.

## maskc—Modify Program Status Word (PSW) Mask Bits

This function modifies indicators in the program status word (PSW) to enable or disable selected interrupts.

### Format

```
#include <sysapi.h>
void maskc(enum t_mask_opt maskc_opt, enum t_mask mask);
```

#### maskc\_opt

One of the following step types:

##### **MASKC\_DISABLE**

Turns off the indicator that represents the mask in order to disable the selected interrupt.

##### **MASKC\_ENABLE**

Turns on the indicator that represents the mask in order to enable the selected interrupt.

#### mask

One of the following interrupt types:

##### **MASKC\_EXT**

External interrupts

##### **MASKC\_IO**

Input/output (I/O) interrupts

##### **MASKC\_PER**

Program event recording interrupts.

### Normal Return

Void.

### Error Return

None.

### Programming Considerations

Only one type of interrupt may be enabled or disabled per function call. Modifying combinations of interrupts is not allowed.

### Examples

The following example enables I/O interrupts.

```
#include <sysapi.h>
...
maskc(MASKC_ENABLE, MASKC_IO);
```

### Related Information

None.

## mkdir—Make a Directory

This function creates a directory.

### Format

```
#include <sys/stat.h>
int mkdir(char *pathname, mode_t mode);
```

#### **pathname**

The name of the new directory.

#### **mode**

The access mode for the new directory.

This function creates a new, empty directory called **pathname**. The file permission bits in **mode** are modified by the file creation mask of the process and then used to set the file permission bits of the directory being created. For more information about the file creation mask, see “umask—Set the File Mode Creation Mask” on page 659; for information about the file permission bits, see “chmod—Change the Mode of a File or Directory” on page 32.

If the S\_ISGID mode bit is set OFF in the parent directory, the group ID of the new directory is set to the group ID of the process; otherwise, the group ID of the new directory is set to the group ID of the parent directory.

The `mkdir` function sets the modification time for the new directory. It also sets the modification time for the directory that contains the new directory.

#### TPF deviation from POSIX

The TPF system does not support the `atime` (access time) or `ctime` (change time) time stamp.

If **pathname** names a symbolic link, `mkdir` fails.

### Normal Return

If successful, the `mkdir` function returns a value of 0.

### Error Return

If unsuccessful, the `mkdir` function does not create a directory; it returns a value of `-1` and sets `errno` to one of the following:

<b>EACCES</b>	The process did not have search permission on some component of <b>pathname</b> , or did not have write permission on the parent directory of the directory to be created.
<b>EEXIST</b>	Either the named file refers to a symbolic link or there is already a file or directory with the given <b>pathname</b> .
<b>ELOOP</b>	A loop exists in symbolic links. This error is issued if the number of symbolic links found while resolving the <b>pathname</b> argument is greater than <code>POSIX_SYMLINK_MAX</code> .
<b>EMLINK</b>	The link count of the parent directory has already reached <code>LINK_MAX</code> (defined in the <code>limits.h</code> header file).

**ENAMETOOLONG**

**pathname** is longer than PATH\_MAX characters or some component of **pathname** is longer than NAME\_MAX characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH\_MAX.

**ENOENT**

Some component of **pathname** does not exist or **pathname** is an empty string.

**ENOSPC**

The file system does not have enough space to contain a new directory or the parent directory cannot be extended.

**ENOTDIR**

A component of the **pathname** prefix is not a directory.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

Because the mkdir function requires an update to an existing directory file and the creation of a new directory file, new directories cannot be created in 1052 or UTIL state.

## Examples

The following example creates a new directory.

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

main() {
    char new_dir[]="new.dir";

    if (mkdir(new_dir, S_IRWXU|S_IRGRP|S_IXGRP) != 0)
        perror("mkdir() error");
    else if (chdir(new_dir) != 0)
        perror("first chdir() error");
    else if (chdir("../") != 0)
        perror("second chdir() error");
    else if (rmdir(new_dir) != 0)
        perror("rmdir() error");
    else
        puts("success!");
}
```

## Related Information

- “chdir—Change the Working Directory” on page 30
- “chmod—Change the Mode of a File or Directory” on page 32
- “umask—Set the File Mode Creation Mask” on page 659.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## mkfifo—Make a FIFO Special File

This function creates a FIFO special file. A *FIFO special file* is a file that is typically used to send data from one process to another so that the receiving process reads the data in first-in-first-out (FIFO) format. A FIFO special file is also known as a *named pipe*.

### Format

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

#### **path**

The name of the new file.

#### **mode**

Specifies the file permission bits of the new file.

This function creates a new file called **path**. The file permission bits in **mode** are modified by the file creation mask of the process and then used to set the file permission bits of the pipe being created. For more information about the file creation mask, see “umask—Set the File Mode Creation Mask” on page 659; for information about the file permission bits, see “chmod—Change the Mode of a File or Directory” on page 32.

**Note:** Because you cannot search or execute a FIFO special file, the `mkfifo` function removes the search and execute permission bits if you specify them.

The user ID of the new file is set to the effective user ID of the process. If the `S_ISGID` mode bit is set to OFF in the parent directory, the group ID of the new file is set to the group ID of the process; otherwise, the group ID of the new file is set to the group ID of the parent directory.

When it has completed successfully, the `mkfifo` function updates the `st_mtime` field in the new file and also updates the `st_mtime` field of the directory that contains the new file.

#### **TPF deviation from POSIX**

The TPF system does not support the `atime` (access time) or `ctime` (change time) time stamp.

### Normal Return

A value of 0.

### Error Return

A value of `-1` and `errno` is set to one of the following:

<b>EACCES</b>	Search permission is denied on a component of <b>path</b> , or write permission is denied on the parent directory of the file to be created.
<b>EEXIST</b>	A file by that name already exists.
<b>ELOOP</b>	A loop exists in symbolic links. This error is issued if the number of symbolic links found while resolving the <b>path</b> argument is greater than <code>POSIX_SYMLLOOP</code> .

**ENAMETOOLONG**

**path** is longer than PATH\_MAX characters or some component of **path** is longer than NAME\_MAX characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH\_MAX.

**ENOENT**

A component of **path** was not found or no **path** was specified.

**ENOMEM**

There is not enough storage space available.

**ENOSPC**

The directory or file system that was intended to hold a new file does not have enough space.

**ENOTDIR**

A component of **path** is not a directory.

## Programming Considerations

- The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

Because the mkfifo function requires an update to an existing directory file, new pipes cannot be created in 1052 or UTIL state.

- In a loosely coupled environment, append the processor ID to **path** to make the path name processor unique because the mkfifo function is processor unique but path names are processor shared. Without processor unique path names, programs will see the path name but will not be able to open the pipe. With processor unique path names, each processor gets its own path name based on the common pipe name.

## Examples

The following example uses the mkfifo function to create a FIFO special file called temp.fifo and then writes and reads from the file before closing it.

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

main() {
    char fn[]="temp.fifo";
    char out[20]="FIFO's are fun!", in[20];
    int rfd, wfd;

    if (mkfifo(fn, S_IRWXU) != 0)
        perror("mkfifo() error");
    else {
        if ((rfd = open(fn, O_RDONLY|O_NONBLOCK)) < 0)
            perror("open() error for read end");
        else {
            if ((wfd = open(fn, O_WRONLY)) < 0)
                perror("open() error for write end");
            else {
                if (write(wfd, out, strlen(out)+1) == -1)
                    perror("write() error");
                else if (read(rfd, in, sizeof(in)) == -1)
                    perror("read() error");
                else printf("read '%s' from the FIFO\n", in);
                close(wfd);
            }
            close(rfd);
        }
    }
}
```

## mkfifo

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

## Related Information

- “chmod—Change the Mode of a File or Directory” on page 32
- “close—Close a File” on page 44
- “open—Open a File” on page 380
- “pipe—Create an Unnamed Pipe” on page 391
- “read—Read from a File” on page 410
- “umask—Set the File Mode Creation Mask” on page 659
- “write—Write Data to a File Descriptor” on page 696.

See *TPF Concepts and Structures* for more information about special files.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



## mknod—Make a Character Special File

This function creates a special file.

### Format

```
#include <sys/stat.h>
int mknod(const char *path, mode_t mode, rdev_t dev_identifier)
```

#### **path**

The name of the new node.

#### **mode**

Must be the value `S_IFCHR` (character special file) bitwise *ORed* with access mode.

#### **dev\_identifier**

The major and minor device numbers of the new node. See *TPF Concepts and Structures* for more information about the major and minor device numbers for special files.

This function creates a new character special file with the path name specified in the **path** argument.

The mode argument determines the type of the special file and (along with the file creation mask) the file permissions of the special file. The TPF `mknod` function only supports the character type of special file: `S_IFCHR`. Bits set in the file creation mask are cleared in the file permissions. For more information about these symbols, see “`chmod`—Change the Mode of a File or Directory” on page 32.

The **dev\_identifier** argument contains the major device number and the minor device number of the special file. The major device number identifies a device driver supporting a class of devices or pseudo devices such as input message devices. The minor device number identifies a specific device or pseudo device in the pertinent class.

When it has completed successfully, the `mknod` function updates the `st_mtime` field in the new file. It also updates the `st_mtime` field of the directory that contains the new file.

#### TPF deviation from POSIX

The TPF system does not support the `atime` (access time) or `ctime` (change time) time stamp.

### Normal Return

If successful, the `mknod` function returns a value of 0.

### Error Return

If unsuccessful, the `mknod` function returns a value of `-1` and sets `errno` to one of the following:

- |               |  |
|---------------|--|
| <b>EACCES</b> | Search permission is denied on a component of <b>path</b> , or write permission is denied on the parent directory of the file to be created. |
| <b>EEXIST</b> | A file by that name already exists.  |

## mknod

<b>ELOOP</b>	A loop exists in symbolic links. This error is issued if the number of symbolic links found while resolving the <b>path</b> argument is greater than POSIX_SYMLINK_MAX.
<b>EMLINK</b>	The link count of the parent directory has already reached the maximum defined for the system.
<b>ENAMETOOLONG</b>	<b>path</b> is longer than PATH_MAX characters or some component of <b>path</b> is longer than NAME_MAX characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX.
<b>ENOENT</b>	A component of <b>path</b> was not found or no <b>path</b> was specified.
<b>ENOSPC</b>	The file system does not have enough space to contain a new directory or the parent directory cannot be extended.
<b>ENOTDIR</b>	A component of <b>path</b> is not a directory.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

Because the mknod function requires an update to an existing directory file, new character special files cannot be created in 1052 or UTIL state.

## Examples

The following example provides status information for a file.

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

#define master 0x00070000

main() {
    char fn[]="char.spec";

    if (mknod(fn, S_IFCHR|S_IRUSR|S_IWUSR, master|0x0001) != 0)
        perror("mknod() error");
    else if (unlink(fn) != 0)
        perror("unlink() error");
}
```

## Related Information

- “mkdir—Make a Directory” on page 326
- “open—Open a File” on page 380.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## MQBACK—Back Out a Queue

This function indicates to the queue manager that all MQPUT, MQPUT1, and MQGET function calls that have occurred since the last syncpoint are to be backed out. Messages put as part of a unit of work are deleted; messages retrieved as part of a unit of work are reinstated on the queue. This call is available only for remote client programs. For local TPF MQSeries applications, use the `tx_rollback` function.

### Format

```
#include <cmqc.h>
void      MQBACK (MQHCONN Hconn,
                  PMQLONG pCompCode,
                  PMQLONG pReason);
```

#### Hconn

The connection handle, which represents the connection to the TPF MQSeries queue manager. The value of **Hconn** was returned by a previous MQCONN call.

#### pCompCode

A pointer to the location to store the completion code, which is one of the following:

##### MQCC\_OK

Successfully completed.

##### MQCC\_FAILED

The call failed.

#### pReason

A pointer to the location to store the reason code that qualifies the completion code.

If the completion code is MQCC\_OK, the reason code is MQRC\_NONE, which indicates a normal return.

If the completion code is MQCC\_FAILED, see “Error Return” for the corresponding reason codes.

See *MQSeries Application Programming Reference* and *MQSeries Message Queue Interface Technical Reference* for more information about MQSeries data types and parameters.

### Normal Return

#### MQCC\_OK

Completion code completed successfully.

#### MQRC\_NONE

Reason code completed successfully.

### Error Return

If the completion code is MQCC\_FAILED, the function failed with one of the following reason codes:

#### MQRC\_CONNECTION\_BROKEN

The connection to the queue manager is lost.

#### MQRC\_ENVIRONMENT\_ERROR

The call was not valid in this environment.

## MQBACK

### **MQRC\_HCONN\_ERROR**

The connection handle is not valid.

### **MQRC\_Q\_MGR\_STOPPING**

The queue manager is stopping.

### **MQRC\_STORAGE\_NOT\_AVAILABLE**

There is not enough storage available.

### **MQRC\_UNEXPECTED\_ERROR**

An unexpected error occurred.

## Programming Considerations

If the **Hconn** parameter is not for the local TPF MQSeries queue manager, this MQBACK function call will be sent by TPF MQSeries client support to the remote queue manager for processing. The options supported by the remote queue manager can differ from the options specified for the local TPF MQSeries queue manager.

## Examples

The following example backs out a queue.

```
#include <cmqc.h>

MQLONG  CompCode;           /* completion code      */
MQLONG  Reason;             /* reason code          */
MQHCONN Hconn;              /* Connection Handle     */

/* Hconn from previous MQCONN MQOPEN */
MQBACK (hCon,&CompCode,&Reason);

if(Reason == MQRC_NONE)
    printf("MQBACK successfully issued\n");
```

## Related Information

- “MQCONN—Connect Queue Manager” on page 339
- “MQDISC—Disconnect Queue Manager” on page 341
- “MQOPEN—Open a Queue” on page 355.

## MQCLOSE—Close a Queue

This function closes a queue.

### Format

```
#include <cmqc.h>
void      MQCLOSE(MQHCONN Hconn,
                  PMQHOBJ pHobj,
                  MQLONG Options,
                  PMQLONG pCompCode,
                  PMQLONG pReason);
```

#### Hconn

The connection handle, which represents the connection to the TPF MQSeries queue manager. The value of **Hconn** was returned by a previous MQCONN call.

#### pHobj

A pointer to the location to store the queue handle, which represents the queue. The value of **Hobj** was returned by a previous MQOPEN call.

#### Options

The options that control the action of MQCLOSE. TPF MQSeries queue manager ignores this parameter.

#### pCompCode

A pointer to the location to store the completion code, which is one of the following:

##### MQCC\_OK

Successfully completed.

##### MQCC\_FAILED

The call failed.

#### pReason

A pointer to the location to store the reason code that qualifies the completion code.

If the completion code is MQCC\_OK, the reason code is MQRC\_NONE, which indicates a normal return.

If the completion code is MQCC\_FAILED, see “Error Return” for the corresponding reason codes.

See *MQSeries Application Programming Reference* and *MQSeries Message Queue Interface Technical Reference* for more information about MQSeries data types and parameters.

### Normal Return

#### MQCC\_OK

Completion code completed successfully.

#### MQRC\_NONE

Reason code completed successfully.

### Error Return

If the completion code is MQCC\_FAILED, the function failed with one of the following reason codes:

#### MQRC\_HCONN\_ERROR

The connection handle is not valid.

## MQCLOSE

### **MQRC\_HOBJ\_ERROR**

The object handle is not valid.

### **MQRC\_Q\_MGR\_NOT\_ACTIVE**

The TPF MQSeries queue manager is not started.

### **MQRC\_Q\_MGR\_NOT\_AVAILABLE**

The TPF MQSeries queue manager is not loaded.

## Programming Considerations

If the **Hconn** parameter is not for the local TPF MQSeries queue manager, this MQCLOSE function call will be sent by TPF MQSeries client support to the remote queue manager for processing. The options supported by the remote queue manager can differ from the options specified for the local TPF MQSeries queue manager.

## Examples

The following example closes a queue.

```
#include <cmqc.h>

PMQHOBj pHobj;
MQLONG  CompCode;           /* completion code      */
MQLONG  Reason;             /* reason code          */
MQHCONN Hcon;               /* Connection Handle     */
MQLONG  C_options = 0;

/* Hcon and Hobj from previous MQCONN MQOPEN */
MQCLOSE(Hcon, pHobj, C_options, &CompCode, &Reason);

if(Reason != MQRC_NONE)
    printf("MQCLOSE ended with reason code %d.\n",Reason);
```

## Related Information

- “MQCONN—Connect Queue Manager” on page 339
- “MQDISC—Disconnect Queue Manager” on page 341
- “MQOPEN—Open a Queue” on page 355.

## MQCMIT–Commit a Queue

This function indicates to the queue manager that the application has reached a syncpoint, and that all MQPUT, MQPUT1, and MQGET function calls that have occurred since the last syncpoint are to be made permanent. Messages put as part of a unit of work are made available to other applications; messages retrieved as part of a unit of work are deleted. This call is available only for remote client programs. For local TPF MQSeries applications, use `tx_commit`.

### Format

```
#include <cmqc.h>
void      MQCMIT (MQHCONN Hconn,
                  PMQLONG pCompCode,
                  PMQLONG pReason);
```

#### Hconn

The connection handle, which represents the connection to the TPF MQSeries queue manager. The value of **Hconn** was returned by a previous MQCONN call.

#### pCompCode

A pointer to the location to store the completion code, which is one of the following:

##### MQCC\_OK

Successfully completed.

##### MQCC\_WARNING

Partially completed.

##### MQCC\_FAILED

The call failed.

#### pReason

A pointer to the location to store the reason code that qualifies the completion code.

If the completion code is MQCC\_OK, the reason code is MQRC\_NONE, which indicates a normal return.

If the completion code is MQCC\_WARNING, see “Error Return” for the corresponding reason codes.

If the completion code is MQCC\_FAILED, see “Error Return” for the corresponding reason codes.

See *MQSeries Application Programming Reference* and *MQSeries Message Queue Interface Technical Reference* for more information about MQSeries data types and parameters.

### Normal Return

#### MQCC\_OK

Completion code completed successfully.

#### MQRC\_NONE

Reason code completed successfully.

### Error Return

If the completion code is MQCC\_WARNING or MQCC\_FAILED, the function failed with one of the following reason codes:

## MQCMIT

### **MQRC\_BACKED\_OUT**

A severe error occurred during a unit of work or the unit of work was backed out.

### **MQRC\_CONNECTION\_BROKEN**

The connection to the queue manager is lost.

### **MQRC\_HCONN\_ERROR**

The connection handle is not valid.

### **MQRC\_OBJECT\_DAMAGED**

An object is damaged.

### **MQRC\_Q\_MGR\_STOPPING**

The queue manager is stopping.

### **MQRC\_STORAGE\_NOT\_AVAILABLE**

There is not enough storage available.

### **MQRC\_UNEXPECTED\_ERROR**

An unexpected error occurred.

## Programming Considerations

If the **Hconn** parameter is not for the local TPF MQSeries queue manager, this MQCMIT function call will be sent by TPF MQSeries client support to the remote queue manager for processing. The options supported by the remote queue manager can differ from the options specified for the local TPF MQSeries queue manager.

## Examples

The following example commits a unit of work.

```
#include <cmqc.h>

MQLONG  CompCode;           /* completion code      */
MQLONG  Reason;             /* reason code          */
MQHCONN Hconn;              /* Connection Handle     */

/* Hconn from previous MQCONN MQOPEN */
MQCMIT(hCon,&CompCode,&Reason);

if(Reason == MQRC_NONE)
    printf("Unit of work successfully completed\n");
```

## Related Information

- “MQCONN—Connect Queue Manager” on page 339
- “MQDISC—Disconnect Queue Manager” on page 341
- “MQOPEN—Open a Queue” on page 355.



## MQCONN—Connect Queue Manager

This function connects an application program to a message queuing queue manager. This function provides a queue manager connection handle, which is used by the application on subsequent message queuing calls.

### Format

```
#include <cmqc.h>
void      MQCONN(PMQCHAR pName,
                 PMQHCONN pHconn,
                 PMQLONG pCompCode,
                 PMQLONG pReason);
```

#### pName

A pointer to the name of a queue manager. If the name consists entirely of blanks, the name of the local TPF MQSeries queue manager is used. If the name does not consist entirely of blanks and is not the name of the local TPF MQSeries queue manager (as defined in the ZMQSC DEF MQP command), the application is connected to a remote queue manager by TPF MQSeries client support.

#### pHconn

A pointer to the location to store the connection handle, which represents the connection to the TPF MQSeries queue manager. You must specify the connection handle on all subsequent message queueing calls issued by the application.

#### pCompCode

A pointer to the location to store the completion code, which is one of the following:

##### **MQCC\_OK**

Successfully completed.

##### **MQCC\_FAILED**

The call failed.

#### pReason

A pointer to the location to store the reason code that qualifies the completion code.

If the completion code is MQCC\_OK, the reason code is MQRC\_NONE, which indicates a normal return.

If the completion code is MQCC\_FAILED, see “Error Return” on page 340 for the corresponding reason codes.

See *MQSeries Application Programming Reference* and *MQSeries Message Queue Interface Technical Reference* for more information about MQSeries data types and parameters.

### Normal Return

#### **MQCC\_OK**

Completion code completed successfully.

#### **MQRC\_NONE**

Reason code completed successfully.

## MQCONN

### Error Return

If the completion code is MQCC\_FAILED, the function failed with one of the following reason codes:

#### **MQRC\_HCONN\_ERROR**

The connection handle is not valid.

#### **MQRC\_Q\_MGR\_NOT\_AVAILABLE**

The queue manager is not available.

#### **MQRC\_Q\_MGR\_NAME\_ERROR**

The queue manager name is not valid.

#### **MQRC\_Q\_MGR\_NOT\_ACTIVE**

The queue manager is not started.

#### **MQRC\_Q\_MGR\_STOPPING**

The queue manager is stopping.

### Programming Considerations

- If the application connects to a remote queue manager (not the local TPF MQSeries queue manager), only the MQCONN function, as described in the *MQSeries Application Programming Guide*, can be used.
- If the application connects to the local TPF MQSeries queue manager, only the MQCONN function, as described in this publication, *TPF C/C++ Language Support User's Guide*, can be used.
- Use the MQDISC function to disconnect from the TPF MQSeries queue manager.

### Examples

The following example connects an application program to the local TPF MQSeries queue manager.

```
#include <cmqc.h>

MQLONG  CompCode;           /* completion code      */
MQLONG  Reason;             /* reason code          */
MQHCONN Hcon = MQHC_UNUSABLE_HCONN; /* Connection Handle    */
CHAR QMgrName [] = "TPF.QMGR";

/*****
/*CONNECT TO QUEUE MANAGER
*****/

MQCONN(QMgrName, &Hcon, &CompCode, &Reason);

if(CompCode == MQCC_FAILED)
{
    printf("MQCONN ended with the reason code %d.\n", Reason);
    exit(Reason);
}
:
```

### Related Information

- “MQDISC–Disconnect Queue Manager” on page 341
- “MQOPEN–Open a Queue” on page 355.

## MQDISC–Disconnect Queue Manager

This function disconnects the application from a TPF MQSeries queue manager.

### Format

```
#include <cmqc.h>
void      MQDISC(PMQHCONN pHconn,
                 PMQLONG pCompCode,
                 PMQLONG pReason);
```

#### **pHconn**

A pointer to the location to store the connection handle, which represents the connection to the TPF MQSeries queue manager. You must specify the connection handle on all subsequent message queuing calls issued by the application. The value of **pHconn** was returned by a previous MQCONN call.

#### **pCompCode**

A pointer to the location to store the completion code, which is one of the following:

##### **MQCC\_OK**

Successfully completed.

##### **MQCC\_FAILED**

The call failed.

#### **pReason**

A pointer to the location to store the reason code that qualifies the completion code.

If the completion code is MQCC\_OK, the reason code is MQRC\_NONE, which indicates a normal return.

If the completion code is MQCC\_FAILED, see “Error Return” for the corresponding reason codes.

See *MQSeries Application Programming Reference* and *MQSeries Message Queue Interface Technical Reference* for more information about MQSeries data types and parameters.

### Normal Return

#### **MQCC\_OK**

Completion code completed successfully.

#### **MQRC\_NONE**

Reason code completed successfully.

### Error Return

If the completion code is MQCC\_FAILED, the function failed with the following reason code:

#### **MQRC\_HCONN\_ERROR**

The connection handle is not valid.

### Programming Considerations

If the **Hconn** parameter is not for a local TPF MQSeries queue manager, this MQDISC function call will be sent by TPF MQSeries client support to the remote

## MQDISC

queue manager for processing. The options supported by the remote queue manager can differ from the options specified for the local TPF MQSeries queue manager.

## Examples

The following example disconnects the application from the queue manager.

```
#include <cmqc.h>

MQLONG  CompCode;           /* completion code      */
MQLONG  Reason;             /* reason code          */
MQHCONN Hcon;               /* Connection Handle     */
:

/*****
/*DISCONNECT FROM QUEUE MANAGER
*****/

/* Hcon from previous MQCONN
   MQDISC(&Hcon, &CompCode, &Reason);

   if(Reason != MQRC_NONE)
       printf("MQDISC ended with reason code %d.\n",Reason);
   :
   :
```

## Related Information

- “MQCONN—Connect Queue Manager” on page 339
- “MQCLOSE—Close a Queue” on page 335.

## MQGET–Get Message from an Open Queue

This function gets a message from a local queue that was opened for input using the MQOPEN function.

### Format

```
#include <cmqc.h>
void      MQGET(MQHCONN Hconn,
                MQHOBJ Hobj,
                PMQVOID pMsgDesc,
                PMQVOID pGetMsgOpts,
                MQLONG BufferLength,
                PMQVOID pBuffer,
                PMQLONG pDataLength,
                PMQLONG pCompCode,
                PMQLONG pReason);
```

#### Hconn

The connection handle, which represents the connection to the TPF MQSeries queue manager. The value of **Hconn** was returned by a previous MQCONN call.

#### Hobj

The object handle, which represents the queue to which a message is to be read. The value of **Hobj** was returned by a previous MQOPEN function call with the MQOO\_INPUT\_AS\_Q\_DEF option specified.

#### pMsgDesc (MQMD)

A pointer to where the message queuing message descriptor (MQMD) structure will be stored. This structure describes the attributes of the message being retrieved from the queue.

The following are the fields in the message queuing message descriptor (MQMD):

#### StrucId (MQCHAR4)

The type of structure, which must be MQMD\_STRUC\_ID. This is an input parameter.

#### Version (MQLONG)

The version number of the structure, which must be MQMD\_VERSION\_1. This is an input parameter.

#### MsgType (MQLONG)

The type of message. This is returned as an output parameter.

The following values are returned:

#### MQMT\_DATAGRAM

The message does not require a reply.

#### MQMT\_REQUEST

The message requires a reply. The name of the queue to which the reply should be sent will be specified in the **ReplyToQ** field.

#### MQMT\_REPLY

The message is a reply to an earlier request message (MQMT\_REQUEST). The message should be sent to the queue indicated by the **ReplyToQ** field of the request message.

**Note:** The TPF MQSeries queue manager does not enforce the request-reply relationship; this is an application responsibility.

## MQGET

### Persistence (MQLONG)

Message persistence. This is returned as an output parameter.

Valid values returned are MQPER\_PERSISTENT and MQPER\_NOT\_PERSISTENT.

### ReplyToQ (MQCHAR48)

The name of the queue which the reply should be sent to is returned in this field when the **MsgType** parameter is MQMT\_REQUEST.

### ReplyToQMgr (MQCHAR48)

The name of the queue manager which the reply should be sent to is returned in this field when the **MsgType** parameter is MQMT\_REQUEST.

### PutApplType (MQLONG)

The type of application that put the message on the queue is returned in this field (MQAT\_TPF for TPF applications).

### PutDate (MQCHAR8)

The date when the message was put on the queue is returned in this field, where:

**YYYY** Year (4 numeric digits)

**MM** Month of year (01 to 12)

**DD** Day of month (01 to 31).

### PutTime (MQCHAR8)

The time when the message was put on the queue. HHMMSSSTH is set by the TPF MQSeries queue manager, where:

**HH** Hours (00 to 23)

**MM** Minutes (00 to 59)

**SS** Seconds (00 to 59)

**T** Tenths of a second (0 to 9)

**H** Hundredths of a second (0 to 9).

**Note:** For all other fields in the MQMD, the TPF MQSeries queue manager does not process the field. The values filled in by the originating application are passed to the destination queue.

### pGetMsgOpts (MQGMO)

A pointer to the message queuing get message options (GMO) structure, which contains the options that control the action of the MQGET function.

### StrucId (MQCHAR4)

The type of structure, which must be MQGMO\_STRUC\_ID.

### Version (MQLONG)

The version number of the structure, which must be MQGMO\_VERSION\_1.

### Options (MQLONG)

The options that control the action of the MQGET function:

#### MQGMO\_WAIT

Wait for the message to arrive. When this option is specified in the MQGMO structure, the MQGET function is suspended until a message arrives. If a message is available in queue, the MQGET function returns to the application immediately. Use the WaitInterval field of the MQGMO structure to specify the maximum time (in milliseconds) that you want

an MQGET function call to wait for a message to arrive on a queue. If no suitable message has arrived after this time has elapsed, the call is completed with MQCC\_FAILED and reason code MQRC\_NO\_MSG\_AVAILABLE. You can specify an unlimited wait interval using the MQWI\_UNLIMITED constant in the WaitInterval field.

#### **MQGMO\_NO\_WAIT**

Return immediately if no message is in the queue. This is the default if MQGMO\_WAIT or MQGMO\_NO\_WAIT is not specified.

#### **MQGMO\_SYNCPOINT**

Get a message with syncpoint control. The request is to operate within the normal unit-of-work protocols. The message is marked as being unavailable to other applications, but it is deleted from the queue only when the unit of work is committed. The message is made available again if the unit of work is backed out. If an application already holds a commit scope, the MQGMO\_SYNCPOINT option does not change the unit of work. When an application does not hold a commit scope, the MQGMO\_SYNCPOINT option opens a new commit scope. This option applies to MQSeries client calls only. An implicit syncpoint occurs for any uncommitted requests when an MQDISC function call is issued from a server.

#### **MQGMO\_NO\_SYNCPOINT**

Get a message without syncpoint control. The request is to operate outside the normal unit of work. The message is deleted from the queue immediately. The message cannot be made available again by backing out the unit of work. This option applies to MQSeries client calls only.

#### **MQGMO\_ACCEPT\_TRUNCATED\_MSG**

Allow truncation of message data.

#### **WaitInterval (MQLONG)**

Wait interval in milliseconds. Specify a positive number or the constant MQWI\_UNLIMITED in the WaitInterval field. This is the approximate time that an MQGET function call waits for a suitable message to arrive. If no suitable message has arrived after this time has elapsed, the call is completed with MQCC\_FAILED and reason code MQRC\_NO\_MSG\_AVAILABLE.

#### **BufferLength**

The length, in bytes, of the buffer area supplied to the function.

#### **pBuffer**

A pointer to the area where the message data will be copied. If **BufferLength** is less than the message length, as much of the message as possible is moved into the buffer; this occurs whether **MQGMO\_ACCEPT\_TRUNCATED\_MSG** is specified on the **Options** field of the GMO structure or not.

The character set and encoding of data in the buffer are given by the **CodedCharSetID** and **Encoding** fields in the **pMsgDesc** parameter. If these differ from the values required by the application, the application must convert the message data to the character set and encoding required.

#### **pDataLength**

A pointer to the length of the message, which is a returned value. This is the length in bytes of the application data in the message. If this is greater than the

## MQGET

value of the **BufferLength** parameter, only the **BufferLength** number of bytes are returned in the **Buffer** parameter. If the value is zero, the message does not contain application data.

### **pCompCode**

A pointer to the location to store the completion code, which is one of the following:

#### **MQCC\_OK**

Successfully completed.

#### **MQCC\_WARNING**

The message was truncated.

#### **MQCC\_FAILED**

The call failed.

### **pReason**

A pointer to the location to store the reason code that qualifies the completion code.

If the completion code is **MQCC\_OK**, the reason code is **MQRC\_NONE**, which indicates a normal return.

If the completion code is **MQCC\_WARNING** or **MQCC\_FAILED**, see “Error Return” for the corresponding reason codes.

See *MQSeries Application Programming Reference* and *MQSeries Message Queue Interface Technical Reference* for more information about MQSeries data types and parameters.

## Normal Return

### **MQCC\_OK**

Completion code completed successfully.

### **MQRC\_NONE**

Reason code completed successfully.

## Error Return

If the completion code is **MQCC\_WARNING**, the reason code is dependent on the value of **MQGMO\_ACCEPT\_TRUNCATED\_MSG** and is one of the following:

### **MQRC\_TRUNCATED\_MSG\_ACCEPTED**

The truncated message is accepted.

### **MQRC\_TRUNCATED\_MSG\_FAILED**

The truncated message failed.

If the completion code is **MQCC\_FAILED**, the function failed with one of the following reason codes:

### **MQRC\_BUFFER\_ERROR**

The buffer parameter is not valid.

### **MQRC\_BUFFER\_LENGTH\_ERROR**

The buffer length parameter is not valid.

### **MQRC\_DATA\_LENGTH\_ERROR**

The data length parameter is NULL.

### **MQRC\_GET\_INHIBITED**

Get is inhibited for the queue.



**MQRC\_GMO\_ERROR**

Get message options structure is not valid.

**MQRC\_HCONN\_ERROR**

The connection handle is not valid.

**MQRC\_HOBJ\_ERROR**

The object handle is not valid.

**MQRC\_MD\_ERROR**

The message descriptor is not valid.

**MQRC\_NO\_MSG\_AVAILABLE**

No message is available.

**MQRC\_NOT\_OPEN\_FOR\_INPUT**

The queue is not open for input.

**MQRC\_OBJECT\_DAMAGED**

The object is damaged.

**MQRC\_OPTIONS\_ERROR**

The options are not valid or are not consistent.

**MQRC\_Q\_DELETED**

The queue has been deleted.

**MQRC\_Q\_MGR\_NOT\_ACTIVE**

The queue manager is not active.

**MQRC\_Q\_MGR\_NOT\_AVAILABLE**

The queue manager is not available.

**MQRC\_STORAGE\_NOT\_AVAILABLE**

There is not enough storage available.

**MQRC\_UNEXPECTED\_ERROR**

An unexpected error occurred.

**MQRC\_WAIT\_INTERVAL\_ERROR**

The wait interval specified in the WaitInterval field is not correct.

## Programming Considerations

- If the **Hconn** parameter is not for a local TPF MQSeries queue manager, this MQGET function call will be sent by TPF MQSeries client support to the remote queue manager for processing. The options supported by the remote queue manager can differ from the options specified for the local TPF MQSeries queue manager.
- The retrieved message is deleted from the queue. The deletion will occur as part of the MQGET function call except for completion code MQCC\_FAILED or reason code MQRC\_TRUNCATED\_MSG\_FAILED, or as part of a TX\_COMMIT function call if the MQGET function is called from a transaction scope.
- When the MQGET function call occurs in a commit scope, the message is marked as being unavailable to other applications, but it is deleted from the queue only when the unit of work is committed. The message is made available again if the unit of work is rolled back.

## Examples

The following example gets a message from a queue.

## MQGET

```
#include <cmqc.h>

MQHOBJ Hobj;
MQLONG Msglen;
MQBYTE *buffer;
MQLONG CompCode;           /* completion code */
MQLONG Reason;             /* reason code */
MQHCONN Hcon;              /* Connection Handle */
MQGMO gmo = {MQGMO_DEFAULT}; /* Get Msg Options */
MQLONG bufLength = 100;
MQMD md = {MQMD_DEFAULT};
:

/*****
/*GET MESSAGE FROM THE QUEUE */
*****/

buffer = (MQBYTE *) calloc(1,bufLength);

/* Hcon and Hobj from previous MQCONN MQOPEN */
MQGET(Hcon, Hobj, &md, &gmo, bufLength,
      buffer, &Msglen, &CompCode, &Reason);

if(Reason != MQRC_NONE && Reason != MQRC_TRUNCATED_MSG_ACCEPTED)
{
    printf("MQGET ended with reason code %d.\n",Reason);
}

else
{
    :
};
free(buffer);
```

## Related Information

- “MQPUT–Put a Message on an Open Queue” on page 359
- “MQOPEN–Open a Queue” on page 355.

## MQINQ—Inquire about Object Attributes

This function returns an array of integers and a set of character strings containing the attributes of a queue or queue manager.

### Format

```
#include <cmqc.h>
void      MQINQ(MQHCONN Hconn,
               MQHOBJ Hobj,
               MQLONG SelectorCount,
               PMQLONG pSelectors,
               MQLONG IntAttrCount,
               PMQLONG pIntAttrs,
               MQLONG CharAttrLength,
               PMQCHAR pCharAttrs,
               PMQLONG pCompCode,
               PMQLONG pReason);
```

#### Hconn

The connection handle, which represents the connection to the TPF MQSeries queue manager. The value of **Hconn** was returned by a previous MQCONN call.

#### Hobj

The object handle, which represents the queue or queue manager for which the attributes are required. The value of **Hobj** was returned by a previous MQOPEN function call with the MQOO\_INQUIRE option specified.

#### SelectorCount

The number of attributes that are to be returned. Specify a value from 0 to 256.

#### pSelectors

A pointer to an array of **SelectorCount** attribute selectors. Each selector identifies an integer or character attribute whose value is required.

You can specify the selectors in any order. Attribute values that correspond to integer attribute selectors (MQIA selectors) are returned in **pIntAttrs** in the same order that they are specified in **pSelectors**. Attribute values that correspond to character attribute selectors (MQCA selectors) are returned in **pCharAttrs** in the same order that they are specified in **pSelectors**. Integer and character selectors can be interleaved; only the relative order in each type is important.

If all of the integer selectors occur first, you can use the same element numbers to address corresponding elements in the **pSelectors** and **pIntAttrs** arrays.

If the **SelectorCount** parameter is zero, **pSelectors** is not used; for this condition, the parameter address passed by programs can be NULL.

The following are selectors for the queue manager:

#### MQCA\_DEAD\_LETTER\_Q\_NAME

The name of the dead-letter queue. MQ\_Q\_NAME\_LENGTH defines the length (in bytes) of the resulting string in **pCharAttrs**.

#### MQCA\_Q\_MGR\_NAME

The name of the local queue manager. MQ\_Q\_MGR\_NAME\_LENGTH defines the length (in bytes) of the resulting string in **pCharAttrs**.

#### MQIA\_CODED\_CHAR\_SET\_ID

The coded character set identifier.

#### MQIA\_MAX\_MSG\_LENGTH

The maximum message length (in bytes).

## MQINQ

### **MQIA\_PLATFORM**

The platform on which the queue manager resides. This is always set to MQPL\_TPF.

The following are selectors for all types of queues:

### **MQIA\_Q\_TYPE**

The queue type.

### **MQIA\_INHIBIT\_PUT**

Indicates if messages can be added to the queue.

The following are selectors for local queues:

### **MQCA\_CREATION\_DATE**

The date that the queue was created. MQ\_CREATION\_DATE\_LENGTH defines the length (in bytes) of the resulting string in **pCharAttrs**.

### **MQCA\_CREATION\_TIME**

The time that the queue was created. MQ\_CREATION\_TIME\_LENGTH defines the length (in bytes) of the resulting string in **pCharAttrs**.

### **MQIA\_CURRENT\_Q\_DEPTH**

The number of messages on a queue.

### **MQIA\_INHIBIT\_GET**

Indicates if messages can be retrieved from the queue.

### **MQIA\_MAX\_MSG\_LENGTH**

The maximum message length.

### **MQIA\_MAX\_Q\_DEPTH**

The maximum number of messages allowed on a queue.

### **MQIA\_Q\_DEPTH\_HIGH\_LIMIT**

The maximum number of messages allowed on the queue before a warning is sent to the console.

### **MQIA\_SHAREABILITY**

Indicates whether the queue can be opened by more than one application at a time. This is always set to MQQA\_SHAREABLE for the TPF system.

### **MQIA\_TRIGGER\_CONTROL**

Indicates whether to trigger or not. You can specify this selector only for local normal queues.

### **MQIA\_TRIGGER\_TYPE**

Indicates the type of trigger (FIRST, EVERY, OR NONE).

### **MQIA\_USAGE**

Specifies whether the queue is a normal or a transmission queue.

The following are selectors for local definitions of remote queues:

### **MQCA\_PROCESS\_NAME**

The name of the remote process definition.  
MQ\_PROCESS\_NAME\_LENGTH defines the length (in bytes) of the resulting string in **pCharAttrs**.

### **MQCA\_REMOTE\_Q\_MGR\_NAME**

The name of the remote queue manager. MQ\_Q\_MGR\_NAME\_LENGTH defines the length (in bytes) of the resulting string in **pCharAttrs**.

**MQCA\_REMOTE\_Q\_NAME**

The name of the remote queue. MQ\_Q\_NAME\_LENGTH defines the length (in bytes) of the resulting string in **pCharAttrs**.

**MQCA\_TRIGGER\_DATA**

The name of the remote process definition. MQ\_TRIGGER\_DATA\_LENGTH defines the length (in bytes) of the resulting string in **pCharAttrs**.

**MQCA\_XMIT\_Q\_NAME**

The transmission queue name. MQ\_Q\_NAME\_LENGTH defines the length (in bytes) of the resulting string in **pCharAttrs**.

The following are selectors for alias queues:

**MQCA\_BASE\_Q\_NAME**

The name of the queue to which the alias refers.

MQ\_Q\_MGR\_NAME\_LENGTH defines the length (in bytes) of the resulting string in **pCharAttrs**.

**MQIA\_INHIBIT\_GET**

Indicates if messages can be retrieved from the queue.

**IntAttrCount**

The number of integer attributes. If there are no integer attributes, specify zero.

**plntAttrs**

A pointer to an array of **IntAttrCount** integer attribute values. Integer attribute values are returned in the same order as the integer selectors (MQIA selectors) in the **pSelectors** parameter. If the array contains more elements than the number of integer selectors, the additional elements are unchanged.

If **Hobj** represents a queue, but an attribute selector is not applicable to that type of queue, the value MQIAV\_NOT\_APPLICABLE is returned for the corresponding element in the **plntAttrs** array.

If the **IntAttrCount** or **SelectorCount** parameter is zero, **plntAttrs** is not used; for this condition, the parameter address passed by programs can be NULL.

**CharAttrLength**

The length (in bytes) of the character attributes buffer. Specify a value that is at least the sum of the lengths of the character attributes specified in **pSelectors**. If there are no character attributes, specify zero.

**pCharAttrs**

A pointer to the buffer in which the character attributes are returned. The attributes are concatenated and returned in the same order as the character selectors (MQCA selectors) specified in the **pSelectors** parameter. The length of each attribute string is fixed for each attribute, and the value returned is padded to the right with blanks if necessary. If the buffer is larger than that needed to contain all the requested character attributes (including padding), the bytes beyond the last attribute value returned are unchanged.

If **Hobj** represents a queue, but an attribute selector is not applicable to that type of queue, a character string consisting of all asterisks (\*) is returned as the value of that attribute in **pCharAttrs**.

If the **CharAttrLength** or **SelectorCount** parameter is zero, **pCharAttrs** is not used; for this condition, the parameter address passed by programs can be NULL.

## MQINQ

### **pCompCode**

A pointer to the location to store the completion code, which is one of the following:

#### **MQCC\_OK**

Successfully completed.

#### **MQCC\_WARNING**

Partially completed.

#### **MQCC\_FAILED**

The call failed.

### **pReason**

A pointer to the location to store the reason code that qualifies the completion code.

If the completion code is MQCC\_OK, the reason code is MQRC\_NONE, which indicates a normal return.

If the completion code is MQCC\_WARNING or MQCC\_FAILED, see “Error Return” for the corresponding reason codes.

See *MQSeries Application Programming Reference* and *MQSeries Message Queue Interface Technical Reference* for more information about MQSeries data types and parameters.

## Normal Return

### **MQCC\_OK**

Completion code completed successfully.

### **MQRC\_NONE**

Reason code completed successfully.

## Error Return

If the completion code is MQCC\_WARNING, the function completed partially with one of the following reason codes:

### **MQRC\_CHAR\_ATTRS\_TOO\_SHORT**

There is not enough space for the character attributes.

### **MQRC\_INT\_ATTR\_COUNT\_TOO\_SMALL**

There is not enough space for the integer attributes.

### **MQRC\_SELECTOR\_NOT\_FOR\_TYPE**

The selector type is not applicable to the queue type.

If the completion code is MQCC\_FAILED, the function failed with one of the following reason codes:

### **MQRC\_CHAR\_ATTR\_LENGTH\_ERROR**

The length of the character attributes is not valid.

### **MQRC\_CHAR\_ATTRS\_ERROR**

The character attributes string is not valid.

### **MQRC\_CONNECTION\_BROKEN**

The connection to the queue manager is lost.

### **MQRC\_HCONN\_ERROR**

The connection handle is not valid.

**MQRC\_HOBJ\_ERROR**

The object handle is not valid.

**MQRC\_INT\_ATTR\_COUNT\_ERROR**

The number of integer attributes is not valid.

**MQRC\_INT\_ATTRS\_ARRAY\_ERROR**

The integer attributes array is not valid.

**MQRC\_NOT\_OPEN\_FOR\_INQUIRE**

The queue is not open for the inquiry.

**MQRC\_OBJECT\_DAMAGED**

The object is damaged.

**MQRC\_Q\_DELETED**

The queue has been deleted.

**MQRC\_Q\_MGR\_NAME\_ERROR**

The queue manager name is not valid.

**MQRC\_Q\_MGR\_NOT\_ACTIVE**

The queue manager is not active.

**MQRC\_Q\_MGR\_NOT\_AVAILABLE**

The queue manager is not available for connection.

**MQRC\_SELECTOR\_COUNT\_ERROR**

The number of selectors is not valid.

**MQRC\_SELECTOR\_ERROR**

The attribute selector is not valid.

**MQRC\_SELECTOR\_LIMIT\_EXCEEDED**

The number of selectors is too large.

**MQRC\_STORAGE\_NOT\_AVAILABLE**

There is not enough storage available.

## Programming Considerations

- If the **Hconn** parameter is not for a local TPF MQSeries queue manager, the MQINQ function call will be sent by TPF MQSeries client support to the remote queue manager for processing. The options supported by the remote queue manager can differ from the options specified for the local TPF MQSeries queue manager.
- The values returned are a snapshot of the selected attributes. There is no guarantee that the attributes will not change before the application can use the returned values.
- If the MQINQ function call is directed to an alias queue, the attributes returned are for those of the alias queue, not those of the base queue to which the alias queue refers.
- If a number of attributes are to be requested, and subsequently some of them are to be set using the MQSET function call, it may be convenient to position the attributes that are to be set at the beginning of the selector arrays so that the same arrays (with reduced counts) can be used for the MQSET function.
- If more than one of the warning conditions occur, the reason code returned is the first one in the following list that applies:

MQRC\_SELECTOR\_NOT\_FOR\_TYPE

MQRC\_INT\_ATTR\_COUNT\_TOO\_SMALL

MQRC\_CHAR\_ATTRS\_TOO\_SHORT

## MQINQ

### Examples

The following example requests the queue type, the time and date that the queue was created, and whether messages can be added to the queue.

```
#include <cmqc.h>
MQLONG Selectors[] = {MQIA_Q_TYPE, MQIA_INHIBIT_PUT,
                      MQCA_CREATION_DATE, MQCA_CREATION_TIME};
MQLONG IntAttrs[2];
const MQLONG ChrSize = MQ_CREATION_DATE_LENGTH +
                      MQ_CREATION_TIME_LENGTH;
char ChrAttrs[ChrSize + 1] = {0};
MQINQ(Hconn, Hobj, 4, Selectors, 2, IntAttrs, ChrSize, ChrAttrs,
      &CompCode, &Reason);

if (CompCode == MQCC_OK)
{
    printf("QType=%d\n", IntAttrs[0]);
    printf("InhibitPut=%d\n", IntAttrs[1]);
    printf("ChrAttrs=%s\n", ChrAttrs);
}
```

### Related Information

- “MQOPEN—Open a Queue” on page 355
- “MQSET—Set Object Attributes” on page 372.



## MQOPEN—Open a Queue

This function permits an application to read messages from, and write messages to, a queue. In addition, this function permits an application to call an MQINQ function for the local queue manager.

### Format

```
#include <cmqc.h>
void      MQOPEN(MQHCONN Hconn,
                 PMQVOID pObjDesc,
                 MQLONG Options,
                 PMQHOBJS pHobj,
                 PMQLONG pCompCode,
                 PMQLONG pReason);
```

#### Hconn

The connection handle, which represents the connection to the TPF MQSeries queue manager. The value of **Hconn** was returned by a previous MQCONN call.

#### pObjDesc

A pointer to the queue or queue manager descriptor. This structure identifies the queue or queue manager to be opened.

The following are the fields in the message queuing object descriptor (MQOD):

#### StruId (MQCHAR4)

The type of structure, which must be MQOD\_STRUC\_ID. This is an output parameter.

#### Version (MQLONG)

The version number of the structure, which must be MQOD\_VERSION\_1. This is an output parameter.

#### ObjectType (MQLONG)

The object type, which must be MQOT\_Q\_MGR or MQOT\_Q.

#### ObjectName (MQCHAR48)

The local name of the object as defined on the TPF MQSeries queue manager identified by **ObjectQMgrName**.

The name must not contain leading or embedded blanks, but may contain trailing blanks; the first NULL character and characters following it are handled as blanks.

If **ObjectType** is MQOT\_Q\_MGR, this field must be all blanks up to the first NULL character or the end of the field.

See the *MQSeries Application Programming Guide* for more information about names.

#### ObjectQMgrName (MQCHAR48)

The name of the TPF MQSeries queue manager on which the **ObjectName** object is defined.

If the name is specified, it must not contain leading or embedded blanks, but may contain trailing blanks; the first null character and characters following it are handled as blanks.

A name that is entirely blank up to the first null character or the end of the field denotes the queue manager to which the application is connected.

If **ObjectType** is MQOT\_Q\_MGR, the name of the local queue manager must be specified explicitly or specified as a blank.

## MQOPEN

### Options

The options controlling the action of the MQOPEN function.

At least one of the following must be specified:

#### MQOO\_INQUIRE

Open the object to retrieve attributes. Specify **only** this value if **ObjectType** is MQOT\_Q\_MGR.

#### MQOO\_SET

Open the queue to set attributes.

#### MQOO\_INPUT\_AS\_Q\_DEF

The application can get messages from a queue.

#### MQOO\_OUTPUT

The application can put messages on a queue.

If two or more options are specified, the values can be either of the following:

- Added together (do not use the same constant more than once).
- Combined using the bitwise OR operation.

### pHobj

A pointer to the location to store the object handle, which represents the queue. The object handle must be specified on subsequent message queuing calls that operate on the queue.

### pCompCode

A pointer to the location to store the completion code, which is one of the following:

#### MQCC\_OK

Successfully completed.

#### MQCC\_FAILED

The call failed.

### pReason

A pointer to the location to store the reason code that qualifies the completion code.

If the completion code is MQCC\_OK, the reason code is MQRC\_NONE, which indicates a normal return.

If the completion code is MQCC\_FAILED, see “Error Return” for the corresponding reason codes.

See *MQSeries Application Programming Reference* and *MQSeries Message Queue Interface Technical Reference* for more information about MQSeries data types and parameters.

## Normal Return

### MQCC\_OK

Completion code completed successfully.

### MQRC\_NONE

Reason code completed successfully.

## Error Return

If the completion code is MQCC\_FAILED, the function failed with one of the following reason codes:

**MQRC\_ALIAS\_BASE\_Q\_TYPE\_ERROR**

The alias base queue is not a remote queue or local queue.

**MQRC\_HCONN\_ERROR**

The connection handle is not valid.

**MQRC\_NAME\_NOT\_VALID\_FOR\_TYPE**

The object type is MQOT\_Q\_MGR but **ObjectName** is not blank.

**MQRC\_OBJECT\_DAMAGED**

The object is damaged.

**MQRC\_OBJECT\_TYPE\_ERROR**

The object type is not MQOT\_Q\_MGR or MQOT\_Q.

**MQRC\_OD\_ERROR**

The object descriptor structure is not valid.

**MQRC\_OPTIONS\_ERROR**

The options are not valid or are not consistent.

**MQRC\_OPTION\_NOT\_VALID\_FOR\_TYPE**

The specified option is not valid for the object type.

**MQRC\_Q\_MGR\_NOT\_ACTIVE**

The queue manager is not active.

**MQRC\_Q\_MGR\_NOT\_AVAILABLE**

The queue manager is not available.

**MQRC\_STORAGE\_NOT\_AVAILABLE**

There is not enough storage available.

**MQRC\_UNKNOWN\_ALIAS\_BASE\_Q**

The alias base queue is not defined.

**MQRC\_UNKNOWN\_OBJECT\_NAME**

The object name is not known.

**MQRC\_UNKNOWN\_OBJECT\_Q\_MGR**

The object type is MQOT\_Q\_MGR but **ObjectQMgrName** is neither blank nor the name of the local queue manager.

**MQRC\_UNKNOWN\_REMOTE\_Q\_MGR**

The remote queue manager is not defined.

**MQRC\_UNKNOWN\_XMIT\_Q**

The transmission queue that is specified in the remote queue definition does not exist.

**MQRC\_XMIT\_Q\_TYPE\_ERROR**

The transmission queue that is specified in the remote queue definition is not a local queue.

**MQRC\_XMIT\_Q\_USAGE\_ERROR**

The transmission queue that is specified in the remote queue definition is not a transmission queue.

## Programming Considerations

- If the **Hconn** parameter is not for a local TPF MQSeries queue manager, this MQOPEN function call will be sent by TPF MQSeries client support to the remote queue manager for processing. The options supported by the remote queue manager can differ from the options specified for the local TPF MQSeries queue manager.

## MQOPEN

- An application can open the same queue or queue manager many times.
- An application can open more than one queue.

## Examples

The following example opens a queue.

```
#include <cmqc.h>
MQLONG  O_options;           /* MQOPEN options      */
MQLONG  OpenCode;           /* MQOPEN completion code */
MQLONG  Reason;             /* reason code         */
MQHCONN Hcon;               /* Connection Handle    */
MQHOBJ  Hobj;               /* Object Handle        */
MQOD    od = {MQOD_DEFAULT}; /* Object Descriptor    */

O_options = MQOO_OUTPUT;

/*****
 *OPEN THE TARGET QUEUE
 *****/

MQOPEN(Hcon, &od, O_options, &Hobj, &OpenCode, &Reason);

if(Reason != MQRC_NONE)
    printf("MQOPEN ended with reason code %d.\n",Reason);

if(OpenCode == MQCC_FAILED)
    printf("The target queue was not opened.\n");
```

## Related Information

- “MQCLOSE—Close a Queue” on page 335
- “MQGET—Get Message from an Open Queue” on page 343
- “MQINQ—Inquire about Object Attributes” on page 349
- “MQPUT—Put a Message on an Open Queue” on page 359
- “MQSET—Set Object Attributes” on page 372.

## MQPUT–Put a Message on an Open Queue

This function puts a message on a queue that was opened for output using the MQOPEN function call.

### Format

```
#include <cmqc.h>
void      MQPUT(MQHCONN Hconn,
               MQHOBJ Hobj,
               PMQVOID pMsgDesc,
               PMQVOID pPutMsgOpts,
               MQLONG BufferLength,
               PMQVOID pBuffer,
               PMQLONG pCompCode,
               PMQLONG pReason);
```

#### Hconn

The connection handle, which represents the connection to the TPF MQSeries queue manager. The value of **Hconn** was returned by a previous MQCONN function call.

#### Hobj

The object handle, which represents the queue to which a message is added. The value of **Hobj** was returned by a previous MQOPEN function call with the MQOO\_OUTPUT option specified.

#### pMsgDesc (MQMD)

A pointer to the message descriptor. This structure describes the attributes of the message being put to the queue.

The following are the fields in the message queuing message descriptor (MQMD):

##### StrucId (MQCHAR4)

The type of structure, which must be MQMD\_STRUC\_ID. This is an output parameter.

##### Version (MQLONG)

The version number of the structure, which must be MQMD\_VERSION\_1. This is an output parameter.

##### MsgType (MQLONG)

The type of message. This is an output parameter. The TPF MQSeries queue manager verifies that the value is valid.

The following values are valid:

##### MQMT\_DATAGRAM

The message does not require a reply.

##### MQMT\_REQUEST

The message requires a reply. The name of the queue to which the reply should be sent must be specified in the **ReplyToQ** field.

##### MQMT\_REPLY

The message is a reply to an earlier request message (MQMT\_REQUEST). The message should be sent to the queue indicated by the **ReplyToQ** field of the request message.

**Note:** The TPF MQSeries queue manager does not enforce the request-reply relationship; this is an application responsibility.

## MQPUT

### Persistence (MQLONG)

Message persistence.

TPF MQSeries supports MQPER\_PERSISTENCE\_AS\_Q\_DEF (the default), MQPER\_PERSISTENT, and MQPER\_NOT\_PERSISTENT.

### MsgId (MQBYTE24)

The message identifier. This is a byte string that is used to distinguish one message from another. Messages cannot have the same identifier, but the queue manager will allow this. The message identifier is a permanent property of the message and continues across restarts of the queue manager. Because the message identifier is a byte string and not a character string, the message identifier is **not** converted between character sets when the message flows from one queue manager to another.

For the MQPUT and MQPUT1 functions, if MQMI\_NONE or MQPMO\_NEW\_MSG\_ID is specified by the application, the queue manager generates a unique message identifier when the message is placed on a queue, and places it in the message descriptor that is sent with the message. The queue manager also returns this message identifier in the message descriptor belonging to the sending application. The application can use this value to record information about particular messages and to respond to queries from other parts of the application.

The sending application can also specify a particular value for the message identifier other than MQMI\_NONE; this stops the queue manager from generating a unique message identifier. An application that is forwarding a message can use this facility to propagate the message identifier of the original message.

The queue manager does not make any use of this field except to:

- Generate a unique value if requested, as described previously
- Deliver the value to the application that issues the MQGET request for the message.

On return from an MQGET function call, the **MsgId** field is set to the message identifier of the message that is returned (if any).

The following special value may be used:

#### MQMI\_NONE

No message identifier is specified.

The value is binary zero for the length of the field.

For C programming language, the constant MQMI\_NONE\_ARRAY is also defined; this has the same value as MQMI\_NONE, but is an array of characters instead of a string.

### ReplyToQ (MQCHAR48)

The name of the queue which the reply should be sent to is returned in this field when the **MsgType** parameter is MQMT\_REQUEST.

The application must provide the appropriate values.

### ReplyToQMgr (MQCHAR48)

The name of the queue manager which the reply should be sent to is returned in this field when the **MsgType** parameter is MQMT\_REQUEST.

The application must provide the appropriate values.

**PutApplType (MQLONG)**

The type of application that put the message (MQAT\_TPF for TPF applications) on the queue.

**PutDate (MQCHAR8)**

The date when the message was put on the queue. YYYYMMDD is set by the TPF MQSeries queue manager, where:

**YYYY** Year (4 numeric digits)

**MM** Month of year (01 to 12)

**DD** Day of month (01 to 31)

**PutTime (MQCHAR8)**

The time when the message was put on the queue. HHMMSSSTH is set by the TPF MQSeries queue manager, where:

**HH** Hours (00 to 23)

**MM** Minutes (00 to 59)

**SS** Seconds (00 to 59)

**T** Tenths of a second (0 to 9)

**H** Hundredths of a second (0 to 9).

**Note:** For all other fields in the MQMD, the TPF MQSeries queue manager does not process the field. The values filled in by the application are passed to the destination queue.

**pPutMsgOpts**

A pointer to the options controlling the action of the MQPUT function.

The following values are valid:

**StrucId (MQCHAR4)**

The type of structure, which must be MQPMO\_STRUC\_ID.

**Version (MQLONG)**

The version number of the structure, which must be MQPMO\_VERSION\_1.

**Options (MQLONG)**

The options controlling the action of the MQPUT function:

The following values are valid:

**MQPMO\_NONE**

No options are specified.

**MQPMO\_NEW\_MSG\_ID**

Generate a new message identifier in a field in the MQMD.

**Note:** Using this option overwrites the **MsgId** field in the **pMsgDesc** parameter even if the **MsgId** field is not set to MQMI\_NONE.

**MQPMO\_SYNCPOINT**

Put a message with syncpoint control. The request is to operate within the normal unit of work. The message is not visible outside the unit of work until the unit of work is committed. If the unit of work is backed out, the message is deleted. This option applies to MQSeries client calls only.

## MQPUT

**Note:** A unit of work is committed by an MQDISC function call, a tx\_commit function call from a local application, or an MQCMIT function call from the client program over a server connection channel. It is backed out by an MQBACK or tx\_rollback function call.

### MQPMO\_NO\_SYNCPOINT

Put a message without syncpoint control. The request is to operate within the normal unit of work. The message is available immediately and it cannot be deleted by backing out the unit of work. This option applies to MQSeries client calls only.

### ResolvedQMgrName (MQCHAR48)

The TPF MQSeries queue manager puts the original queue manager name in this field. The TPF system does not support an alias name.

### ResolvedQName (MQCHAR48)

The TPF MQSeries queue manager puts the original queue name in this field. The TPF system does not support an alias name.

### BufferLength

The length of the message in the buffer. Specify 0 for a message that has no data.

### pBuffer

A pointer to the message data. This is a buffer containing the application data to be sent. If the **BufferLength** parameter is zero, the **pBuffer** parameter can be NULL.

### pCompCode

A pointer to the location to store the completion code, which is one of the following:

#### MQCC\_OK

Successfully completed.

#### MQCC\_FAILED

The call failed.

### pReason

A pointer to the location to store the reason code that qualifies the completion code.

If the completion code is MQCC\_OK, the reason code is MQRC\_NONE, which indicates a normal return.

If the completion code is MQCC\_FAILED, see “Error Return” on page 363 for the corresponding reason codes.

See *MQSeries Application Programming Reference* and *MQSeries Message Queue Interface Technical Reference* for more information about MQSeries data types and parameters.

## Normal Return

### MQCC\_OK

Completion code completed successfully.

### MQRC\_NONE

Reason code completed successfully.



## Error Return

If the completion code is MQCC\_FAILED, the function failed with one of the following reason codes:

### **MQRC\_BUFFER\_ERROR**

The buffer parameter is not valid.

### **MQRC\_BUFFER\_LENGTH\_ERROR**

The buffer length parameter is not valid.

### **MQRC\_HCONN\_ERROR**

The connection handle is not valid.

### **MQRC\_HOBJ\_ERROR**

The object handle is not valid.

### **MQRC\_MD\_ERROR**

The message descriptor is not valid.

### **MQRC\_MISSING\_REPLY\_TO\_Q**

The reply to the queue is missing.

### **MQRC\_MSG\_TOO\_BIG\_FOR\_Q**

The message length is greater than the maximum for the queue.

### **MQRC\_MSG\_TYPE\_ERROR**

The message type in the message descriptor is not valid.

### **MQRC\_NOT\_OPEN\_FOR\_OUTPUT**

The queue is not open for output.

### **MQRC\_OPTIONS\_ERROR**

The options are not valid or are not consistent.

### **MQRC\_PERSISTENCE\_ERROR**

Persistence is not valid.

### **MQRC\_PMO\_ERROR**

The put message options structure is not valid.

### **MQRC\_PUT\_INHIBITED**

The put calls are inhibited for the queue.

### **MQRC\_Q\_DELETED**

The queue has been deleted.

### **MQRC\_Q\_FULL**

The queue already contains the maximum number of messages.

### **MQRC\_Q\_MGR\_NOT\_ACTIVE**

The queue manager is not active.

### **MQRC\_Q\_MGR\_NOT\_AVAILABLE**

The queue manager is not available.

### **MQRC\_STORAGE\_NOT\_AVAILABLE**

There is not enough storage available.

### **MQRC\_UNEXPECTED\_ERROR**

An unexpected error occurred.

## Programming Considerations

- If the **Hconn** parameter is not for a local TPF queue manager, this MQPUT function call will be sent by TPF MQSeries client support to the remote queue manager

## MQPUT

for processing. The options supported by the remote queue manager can differ from the options specified for the local TPF MQSeries queue manager.

- If the MQPUT function is called from within a TPF commit scope and the commit scope is rolled back, the MQPUT function call is canceled.
- If **ReplyToQMGr** is blank, the TPF MQSeries queue manager will resolve **ReplyToQ** as a local queue, a local definition of a remote queue, or an alias queue.
- If a program has an open commit scope when it uses the MQPUT or MQPUT1 function to put a message on a queue, the message is not visible outside the unit of work until the unit of work is committed. If the unit of work is rolled back, the message is deleted.

## Examples

The following example puts a message on an opened queue.

```
#include <cmqc.h>
```

```
MQBYTE  buffer[] = "THIS IS A MESSAGE";
MQMD    md = {MQMD_DEFAULT};           /* Message Descriptor */
MQPMO   pmo = {MQPMO_DEFAULT};         /* Put Msg Options    */
MQLONG  msgLength = sizeof(buffer);     /* Msg length        */
MQHCONN Hcon;                          /* Connection Handle  */
MQHOBJ  Hobj;                          /* Object Handle      */
MQLONG  CompCode;                      /* completion code    */
MQLONG  Reason;                        /* reason code        */

:
:
md.Persistence = MQPER_PERSISTENT;
md.MsgType     = MQMT_DATAGRAM;

/* Hcon Hobj from previous MQCONN MQOPEN calls */
MQPUT(Hcon, Hobj, &md, &pmo,
      msgLength, buffer, &CompCode, &Reason);

if(Reason != MQRC_NONE)
{
    printf("MQPUT ended with reason code %d\n", Reason);
} else
{
:
}
```

## Related Information

- “MQGET–Get Message from an Open Queue” on page 343
- “MQOPEN–Open a Queue” on page 355.

## MQPUT1—Put a Single Message on a Queue

This function places a single message on a queue; the queue does not have to be open.

### Format

```
#include <cmqc.h>
void MQPUT1(MQHCONN Hconn,
            PMQVOID pObjDesc,
            PMQVOID pMsgDesc,
            PMQVOID pPutMsgOpts,
            MQLONG BufferLength,
            PMQVOID pBuffer,
            PMQLONG pCompCode,
            PMQLONG pReason);
```

#### Hconn

The connection handle, which represents the connection to the TPF MQSeries queue manager. The value of **Hconn** was returned by a previous MQCONN call.

#### pObjDesc

A pointer to the object descriptor. This structure identifies the queue to which the message is added.

#### pMsgDesc (MQMD)

A pointer to the message descriptor. This structure describes the attributes of the message being put on the queue.

The following are the fields in the message queuing message descriptor (MQMD):

##### StrucId (MQCHAR4)

The type of structure, which must be MQMD\_STRUC\_ID.

##### Version (MQLONG)

The version number of the structure, which must be MQMD\_VERSION\_1.

##### MsgType (MQLONG)

The type of message. The TPF MQSeries queue manager verifies that the value is valid.

The following values are valid:

##### MQMT\_DATAGRAM

The message does not require a reply.

##### MQMT\_REQUEST

The message requires a reply. The name of the queue to which the reply should be sent must be specified in the **ReplyToQ** field.

##### MQMT\_REPLY

The message is a reply to an earlier request message (MQMT\_REQUEST). The message should be sent to the queue indicated by the **ReplyToQ** field of the request message.

**Note:** The TPF MQSeries queue manager does not enforce the request-reply relationship; this is an application responsibility.

#### Persistence (MQLONG)

Message persistence.

TPF MQSeries supports MQPER\_PERSISTENT (the default) and MQPER\_NOT\_PERSISTENT.

### **MsgId (MQBYTE24)**

The message identifier. This is a byte string that is used to distinguish one message from another. Messages cannot have the same identifier, but the queue manager will allow this. The message identifier is a permanent property of the message and continues across restarts of the queue manager. Because the message identifier is a byte string and not a character string, the message identifier is **not** converted between character sets when the message flows from one queue manager to another.

For the MQPUT and MQPUT1 functions, if MQMI\_NONE or MQPMO\_NEW\_MSG\_ID is specified by the application, the queue manager generates a unique message identifier when the message is placed on a queue and places it in the message descriptor that is sent with the message. The queue manager also returns this message identifier in the message descriptor belonging to the sending application. The application can use this value to record information about particular messages and to respond to queries from other parts of the application.

The sending application can also specify a particular value for the message identifier other than MQMI\_NONE; this stops the queue manager from generating a unique message identifier. An application that is forwarding a message can use this facility to propagate the message identifier of the original message.

The queue manager does not make any use of this field except to:

- Generate a unique value if requested, as described previously
- Deliver the value to the application that issues the MQGET request for the message.

On return from an MQGET function call, the **MsgId** field is set to the message identifier of the message that is returned (if any).

The following special value may be used:

#### **MQMI\_NONE**

No message identifier is specified.

The value is binary zero for the length of the field.

For C programming language, the constant MQMI\_NONE\_ARRAY is also defined; this has the same value as MQMI\_NONE, but is an array of characters instead of a string.

### **ReplyToQ (MQCHAR48)**

The name of reply to queue when the **MsgType** parameter is MQMT\_REQUEST.

The application must provide the appropriate values.

### **ReplyToQMgr (MQCHAR48)**

The name of the reply to queue manager when the **MsgType** parameter is MQMT\_REQUEST.

The application must provide the appropriate values.

### **PutApplType (MQLONG)**

The type of application that put the message (MQAT\_TPF for TPF applications) on the queue.

**PutDate (MQCHAR8)**

The date when the message was put on the queue. YYYYMMDD is set by the TPF MQSeries queue manager, where:

**YYYY** Year (4 numeric digits)

**MM** Month of year (01 to 12)

**DD** Day of month (01 to 31).

**PutTime (MQCHAR8)**

The time when the message was put on the queue. HHMMSSSTH is set by the TPF MQSeries queue manager, where:

**HH** Hours (00 to 23)

**MM** Minutes (00 to 59)

**SS** Seconds (00 to 59)

**T** Tenths of a second (0 to 9)

**H** Hundredths of a second (0 to 9).

**Note:** For all other fields in the MQMD, the TPF MQSeries queue manager does not process the field. The values filled in by the application are passed to the destination queue.

**pPutMsgOpts**

A pointer to the options controlling the action of the MQPUT1 function.

The following values are valid:

**StrucId (MQCHAR4)**

The type of structure, which must be MQPMO\_STRUC\_ID.

**Version (MQLONG)**

The version number of the structure, which must be MQPMO\_VERSION\_1.

**Options (MQLONG)**

The options controlling the action of the MQPUT1 function:

**MQPMO\_NONE**

No options are specified.

**MQPMO\_NEW\_MSG\_ID**

Generate a new message identifier in a field in the MQMD.

**Note:** Using this option overwrites the **MsgId** field in the **pMsgDesc** parameter even if the **MsgId** field is not set to MQMI\_NONE.

**MQPMO\_SYNCPOINT**

Put a message with syncpoint control. The request is to operate within the normal unit of work. The message is not visible outside the unit of work until the unit of work is committed. If the unit of work is backed out, the message is deleted. This option applies to MQSeries client calls only.

**Note:** A unit of work is committed by an MQDISC function call, a tx\_commit from a local application, or an MQCMIT function call from the client program over a server connection channel. It is backed out by an MQBACK or tx\_rollback function call.

**MQPMO\_NO\_SYNCPOINT**

Put a message without syncpoint control. The request is to operate

## MQPUT1

within the normal unit of work. The message is available immediately and it cannot be deleted by backing out the unit of work. This option applies to MQSeries client calls only.

### **ResolvedQMgrName (MQCHAR48)**

The TPF MQSeries queue manager puts the original queue manager name in this field. The TPF system does not support an alias name.

### **ResolvedQName (MQCHAR48)**

The TPF MQSeries queue manager puts the original queue name in this field. The TPF system does not support an alias name.

### **BufferLength**

The length of the message in the buffer. Specify 0 for a message that has no data.

### **pBuffer**

A pointer to the message data. This is a buffer containing the application data to be sent. If the **BufferLength** parameter is zero, the **pBuffer** parameter can be NULL.

### **pCompCode**

A pointer to the location to store the completion code, which is one of the following:

#### **MQCC\_OK**

Successfully completed.

#### **MQCC\_FAILED**

The call failed.

### **pReason**

A pointer to the location to store the reason code that qualifies the completion code.

If the completion code is MQCC\_OK, the reason code is MQRC\_NONE, which indicates a normal return.

If the completion code is MQCC\_FAILED, see “Error Return” for the corresponding reason codes.

See *MQSeries Application Programming Reference* and *MQSeries Message Queue Interface Technical Reference* for more information about MQSeries data types and parameters.

## Normal Return

### **MQCC\_OK**

Completion code completed successfully.

### **MQRC\_NONE**

Reason code completed successfully.

## Error Return

If the completion code is MQCC\_FAILED, the function failed with one of the following reason codes:

### **MQRC\_ALIAS\_BASE\_Q\_TYPE\_ERROR**

The alias base queue is not a remote queue or local queue.

### **MQRC\_BUFFER\_ERROR**

The buffer parameter is not valid.

**MQRC\_BUFFER\_LENGTH\_ERROR**

The buffer length parameter is not valid.

**MQRC\_HCONN\_ERROR**

The connection handle is not valid.

**MQRC\_HOBJ\_ERROR**

The object handle is not valid.

**MQRC\_MD\_ERROR**

The message descriptor is not valid.

**MQRC\_MISSING\_REPLY\_TO\_Q**

The reply to the queue is missing.

**MQRC\_MSG\_TOO\_BIG\_FOR\_Q**

The message length is greater than the maximum for the queue.

**MQRC\_MESSAGE\_TYPE\_ERROR**

The message type in the message descriptor is not valid.

**MQRC\_NAME\_NOT\_VALID\_FOR\_TYPE**

The object type is MQOT\_Q\_MGR but **ObjectName** is not blank.

**MQRC\_NOT\_OPEN\_FOR\_OUTPUT**

The queue is not open for output.

**MQRC\_OBJECT\_DAMAGED**

The object is damaged.

**MQRC\_OBJECT\_TYPE\_ERROR**

The object type is not MQOT\_Q\_MGR or MQOT\_Q.

**MQRC\_OD\_ERROR**

The object descriptor structure is not valid.

**MQRC\_OPTIONS\_ERROR**

The options are not valid or are not consistent.

**MQRC\_OPTION\_NOT\_VALID\_FOR\_TYPE**

The specified option is not valid for the object type.

**MQRC\_PERSISTENCE\_ERROR**

Persistence is not valid.

**MQRC\_PMO\_ERROR**

The put message options structure is not valid.

**MQRC\_PUT\_INHIBITED**

The put calls are inhibited for the queue.

**MQRC\_Q\_DELETED**

The queue has been deleted.

**MQRC\_Q\_FULL**

The queue already contains the maximum number of messages.

**MQRC\_Q\_MGR\_NOT\_ACTIVE**

The queue manager is not active.

**MQRC\_Q\_MGR\_NOT\_AVAILABLE**

The queue manager is not available.

**MQRC\_STORAGE\_NOT\_AVAILABLE**

There is not enough storage available.

## MQPUT1

### **MQRC\_UNEXPECTED\_ERROR**

An unexpected error occurred.

### **MQRC\_UNKNOWN\_ALIAS\_BASE\_Q**

The alias base queue is not defined.

### **MQRC\_UNKNOWN\_OBJECT\_NAME**

The object name is not known.

### **MQRC\_UNKNOWN\_OBJECT\_Q\_MGR**

The object type is MQOT\_Q\_MGR but **ObjectQMgrName** is neither blank nor the name of the local queue manager.

### **MQRC\_UNKNOWN\_REMOTE\_Q\_MGR**

The remote queue manager is not defined.

### **MQRC\_UNKNOWN\_XMIT\_Q**

The transmission queue that is specified in the remote queue definition does not exist.

### **MQRC\_XMIT\_Q\_TYPE\_ERROR**

The transmission queue that is specified in the remote queue definition is not a local queue.

### **MQRC\_XMIT\_Q\_USAGE\_ERROR**

The transmission queue that is specified in the remote queue definition is not a transmission queue.

## Programming Considerations

- If the **Hconn** parameter is not for a local TPF MQSeries queue manager, this MQPUT1 function call will be sent by TPF MQSeries client support to the remote queue manager for processing. The options supported by the remote queue manager can differ from the options specified for the local TPF MQSeries queue manager.
- If the MQPUT1 function is called from within a TPF commit scope and the commit scope is rolled back, the MQPUT1 function call is canceled.
- If a program has an open commit scope when it uses the MQPUT or MQPUT1 function to put a message on a queue, the message is not visible outside the unit of work until the unit of work is committed. If the unit of work is rolled back, the message is deleted.

## Examples

The following example opens a queue, puts a single message on it, and then closes it.

```
#include <cmqc.h>
MQBYTE buffer[] = "THIS IS A MESSAGE";
MQMD md = {MQMD_DEFAULT};           /* Message Descriptor */
MQPMO pmo = {MQPMO_DEFAULT};        /* Put Msg Options */
MQLONG msgLength = sizeof(buffer);  /* Msg length */
MQHCONN Hcon;                       /* Connection Handle */
MQLONG CompCode;                   /* completion code */
MQLONG Reason;                     /* reason code */
MQOD od = {MQOD_DEFAULT};

:

md.Persistence = MQPER_PERSISTENT;
md.MsgType = MQMT_DATAGRAM;

/* Hcon from previous MQCONN calls */
```



```
MQPUT1(Hcon, &od, &md, &pmo,  
        msgLength, buffer, &CompCode, &Reason);  
  
    if(Reason != MQRC_NONE)  
    {  
        printf("MQPUT ended with reason code %d\n", Reason);  
    } else  
    {  
        :  
    }
```

## Related Information

- “MQGET—Get Message from an Open Queue” on page 343
- “MQOPEN—Open a Queue” on page 355
- “MQPUT—Put a Message on an Open Queue” on page 359.

## MQSET–Set Object Attributes

This function changes the attributes of an MQSeries queue.

### Format

```
#include <cmqc.h>
void      MQSET(MQHCONN Hconn,
               MQHOBJ Hobj,
               MQLONG SelectorCount,
               PMQLONG pSelectors,
               MQLONG IntAttrCount,
               PMQLONG pIntAttrs,
               MQLONG CharAttrLength,
               PMQCHAR pCharAttrs,
               PMQLONG pCompCode,
               PMQLONG pReason);
```

#### Hconn

The connection handle, which represents the connection to the TPF MQSeries queue manager. The value of **Hconn** was returned by a previous MQCONN call.

#### Hobj

The object handle, which represents the queue whose attributes are to be set. The value of **Hobj** was returned by a previous MQOPEN function call with the MQOO\_SET option specified.

#### SelectorCount

The number of attributes that are to be set. Specify a value from 0 to 256.

#### pSelectors

A pointer to an array of **SelectorCount** attribute selectors. Each selector identifies an integer or character attribute whose value is to be set.

You can specify the selectors in any order. Attribute values that correspond to integer attribute selectors (MQIA selectors) must be specified in **pIntAttrs** in the same order that they are specified in **pSelectors**.

If you specify the same selector twice, the last value specified for a given selector is the one that is used.

If the **SelectorCount** parameter is zero, **pSelectors** is not used; for this condition, the parameter address passed by programs can be NULL.

You can specify the following selectors:

#### MQIA\_INHIBIT\_PUT

Specifies if messages can be added to the queue. You can specify this selector for any type of queue.

#### MQIA\_INHIBIT\_GET

Specifies if messages can be retrieved from the queue. You can specify this selector only for local or alias queues.

#### MQIA\_TRIGGER\_CONTROL

Indicates whether to trigger or not. You can specify this selector only for local normal queues.

#### MQIA\_TRIGGER\_TYPE

Indicates the type of trigger (FIRST, EVERY, OR NONE). You can specify this selector only for local normal queues.

#### MQCA\_TRIGGER\_DATA

The name of the remote process definition. MQ\_TRIGGER\_DATA\_LENGTH defines the length (in bytes) of the resulting string in **pCharAttrs**.

**IntAttrCount**

The number of integer attributes. You must specify at least the number of integer attribute selectors that are specified in **pSelectors**. If there are no integer attributes, specify zero.

**pIntAttrs**

A pointer to an array of **IntAttrCount** integer attribute values. Specify these values in the same order as the integer attribute selectors (MQIA selectors) in the **pSelectors** parameter. If the array contains more elements than the number of integer selectors, the additional elements are unchanged.

If the **IntAttrCount** or **SelectorCount** parameter is zero, **pIntAttrs** is not used; for this condition, the parameter address passed by programs can be NULL.

**CharAttrLength**

The length (in bytes) of the character attributes buffer. Specify a value that is at least the sum of the lengths of the character attributes specified in **pSelectors**. If there are no character attributes, specify zero.

**pCharAttrs**

A pointer to the buffer in which the character attributes are returned. The attributes are concatenated and returned in the same order as the character selectors (MQCA selectors) specified in the **pSelectors** parameter. The length of each attribute string is fixed for each attribute, and the value returned is padded to the right with blanks if necessary. If the buffer is larger than that needed to contain all the requested character attributes (including padding), the bytes beyond the last attribute value returned are unchanged.

If **Hobj** represents a queue, but an attribute selector is not applicable to that type of queue, a character string consisting of all asterisks (\*) is returned as the value of that attribute in **pCharAttrs**.

If the **CharAttrLength** or **SelectorCount** parameter is zero, **pCharAttrs** is not used; for this condition, the parameter address passed by programs can be NULL.

**pCompCode**

A pointer to the location to store the completion code, which is one of the following:

**MQCC\_OK**

Successfully completed.

**MQCC\_FAILED**

The call failed.

**pReason**

A pointer to the location to store the reason code that qualifies the completion code.

If the completion code is MQCC\_OK, the reason code is MQRC\_NONE, which indicates a normal return.

If the completion code is MQCC\_FAILED, see "Error Return" on page 374 for the corresponding reason codes.

See *MQSeries Application Programming Reference* and *MQSeries Message Queue Interface Technical Reference* for more information about MQSeries data types and parameters.

## MQSET

### Normal Return

#### **MQCC\_OK**

Completion code completed successfully.

#### **MQRC\_NONE**

Reason code completed successfully.

### Error Return

If the completion code is MQCC\_FAILED, the function failed with one of the following reason codes:

#### **MQRC\_CHAR\_ATTR\_LENGTH\_ERROR**

The length of the character attributes is not valid.

#### **MQRC\_CHAR\_ATTRS\_ERROR**

The character attributes string is not valid.

#### **MQRC\_CONNECTION\_BROKEN**

The connection to the queue manager is lost.

#### **MQRC\_CONNECTION\_STOPPING**

The connection is shutting down.

#### **MQRC\_HCONN\_ERROR**

The connection handle is not valid.

#### **MQRC\_HOBJ\_ERROR**

The object handle is not valid.

#### **MQRC\_INHIBIT\_VALUE\_ERROR**

The value to be set for MQIA\_INHIBIT\_GET or MQIA\_INHIBIT\_PUT is not valid.

#### **MQRC\_INT\_ATTR\_COUNT\_ERROR**

The number of integer attributes is not valid.

#### **MQRC\_INT\_ATTRS\_ARRAY\_ERROR**

The integer attributes array is not valid.

#### **MQRC\_NOT\_OPEN\_FOR\_SET**

The queue is not open for the set request.

#### **MQRC\_Q\_DELETED**

The queue has been deleted.

#### **MQRC\_Q\_MGR\_NAME\_ERROR**

The queue manager name is not valid.

#### **MQRC\_Q\_MGR\_NOT\_AVAILABLE**

The queue manager is not available for connection.

#### **MQRC\_Q\_MGR\_NOT\_ACTIVE**

The queue manager is not active.

#### **MQRC\_Q\_MGR\_STOPPING**

The queue manager is stopping.

#### **MQRC\_SELECTOR\_COUNT\_ERROR**

The number of selectors is not valid.

#### **MQRC\_SELECTOR\_ERROR**

The attribute selector is not valid.

**MQRC\_SELECTOR\_LIMIT\_EXCEEDED**

The number of selectors is too large.

**MQRC\_STORAGE\_NOT\_AVAILABLE**

There is not enough storage available.

**MQRC\_TRIGGER\_CONTROL\_ERROR**

The value of the trigger-control attribute is not valid.

**MQRC\_TRIGGER\_TYPE\_ERROR**

The value of the trigger-type attribute is not valid.

**MQRC\_UNEXPECTED\_ERROR**

An unexpected error occurred.

## Programming Considerations

- If the **Hconn** parameter is not for a local TPF MQSeries queue manager, the MQSET function call will be sent by TPF MQSeries support to the remote queue manager for processing. The options supported by the remote queue manager can differ from the options specified for the local TPF MQSeries queue manager.
- When using this function call, the application program can specify an array of integer attributes. The attributes specified are all set simultaneously if no errors occur. If an error does occur (for example, if a selector is not valid or an attempt is made to set an attribute to a value that is not valid), the call fails and no attributes are set.
- Use the MQINQ function to determine the values of the attributes.

**Note:** There are some attributes whose values you can request with the MQINQ function that cannot be set using the MQSET function. For example, you cannot set any queue manager attributes.

- Attribute changes are preserved across restarts of the queue manager.

## Examples

The following example specifies that messages can be added to the queue.

```
#include <cmqc.h>
MQLONG SelectorsSet[] = {MQIA_INHIBIT_PUT};
MQLONG IntAttrsSet[] = {MQQA_PUT_ALLOWED};

MQSET(Hconn, Hobj, 1, SelectorsSet, 1, IntAttrsSet, 0, NULL,
      &CompCode, &Reason);
```

## Related Information

- “MQOPEN—Open a Queue” on page 355
- “MQINQ—Inquire about Object Attributes” on page 349.

## newCache—Create a New Logical Record Cache

This function creates a new logical record cache.

### Format

```
#include <c$cach.h>
long      newCache (const void *cache_name,
                    cacheToken *cache_token,
                    const long *primary_key_length,
                    const long *secondary_key_length,
                    const long *data_length,
                    const long *number_entries,
                    const long castoutTime,
                    const char *type_of_cache,
                    const long *reserved);
```

#### cache\_name

The name of the logical record cache. The logical record cache name must meet the following requirements:

- The name must be 4 to 12 characters long and padded on the right with blanks if necessary.
- The name must begin with an alphabetic character.
- The name can contain numeric characters, uppercase alphabetic characters, the special characters \$ or @, or underscore (\_).
- The name must be identical on every processor in the complex that is caching the same data.

Logical record cache and coupling facility (CF) cache support use this information to name the logical record cache. See *TPF Application Programming* for more information about logical record cache support and *TPF Database Reference* for more information about CF cache support.

#### cache\_token

The address of the cacheToken field where the assigned value for the cache token will be returned. This value is passed on all future cache access requests for the specified logical record cache. This value enables logical record cache support to uniquely identify the correct logical record cache for the request.

#### primary\_key\_length

The maximum length of the primary key for a cache entry. The value must be from 1 to 256.

#### secondary\_key\_length

The maximum length of the secondary key for a cache entry. The value must be from 1 to 256. If a secondary key will not be used, set this value to zero.

#### data\_length

The maximum length of the data area for a cache entry in the logical record cache. The actual data element may be smaller than this value, but the storage allocation assumes the maximum data length. The value must be from 1 to 4096.

#### number\_entries

The minimum number of unique cache entries the cache will hold before it begins to reuse the least-referenced cache entries. The value must be from 1 to 999 999 999. This value is used if there is no cache entry in the cache control record. If you have defined the logical record cache using the ZCACH command, the number of cache entries defined for the logical record cache in the cache control record is used. In other words, the value specified with the

ZCACH command overrides that specified for the newCache function. See *TPF Operations* for more information about the ZCACH command.

For processor shared entries, this value is ignored if the cache is using a CF cache structure. If the cache must fall back to local mode because of the loss of all CFs, this value will be used as defined previously.

**castoutTime**

The number of seconds, specified as an integer value, that a cache entry can reside in cache before it is cast out. Once this value is reached, the cache entry is marked as *not valid* and results in a *not found* condition. This forces a cache update for the cache entry. If zero is specified, the cache entries are not aged out of cache.

**type\_of\_cache**

A defined value that indicates whether the cache is processor shared or processor unique. Code Cache\_ProcS for processor shared. Code Cache\_ProcQ for processor unique.

**reserved**

Reserved for future use by IBM. Set **reserved** to NULL.

**Normal Return****CACHE\_SUCCESS**

The function is completed successfully.

**Error Return**

One of the following:

**CACHE\_ERROR\_REDEFINE**

The logical record cache already exists. The entry size values on this request are different than the values used to create the logical record cache.

**CACHE\_ERROR\_PARAM**

The value passed for one of the parameters is not valid.

**CACHE\_ERROR\_FULL**

The maximum number of logical record caches that can be created has already been reached.

**CACHE\_ERROR\_GSYS**

Unable to allocate the system heap to fulfill the logical record cache size request.

**Programming Considerations**

- This function uses the contents of the cache control record and the information specified with the ZCACH command to determine the size of the logical record cache you are creating. If the logical record cache was defined previously through the ZCACH command, its specifications override those specified for the newCache function. If the logical record cache has not been defined yet, the newCache function creates the logical record cache using the passed values. See *TPF Operations* for more information about the ZCACH command.
- For processor shared caches using the CF, enter the ZCFCH command to override the logical record cache size. See *TPF Operations* for more information about the ZCFCH command.

## newCache

### Examples

The following example creates a new logical record cache for holding file system directory entries.

```
#include <c$cach.h>

#include <i$glue.h>

struct icontrol * contrl_ptr; /* pointer file system control area */
char   cache_name [CACHE_MAX_NAME_SIZE]; /* cache name build area*/
long   primeKeyLgh=NAME_MAX;      /* max lgh of a path name */
long   secondaryKeyLgh=sizeof(ino_t); /* INODE ordinal numb lgh */
      /* data length for directory cache */
long   dataLgh=sizeof(struct TPF_directory_entry);
      /* invalidate entry after x seconds*/
long   castOutTime = TPF_FS_CACHE_TIMEOUT ;
char   cacheType=Cache_ProcS;    /* processor shared cache      */

memset(cache_name, 0x00, CACHE_MAX_NAME_SIZE);
strcpy(cache_name, TPF_FS_DIR_CACHE_NAME);

contrl_ptr = cinfc_fast_ss(CINFC_CMMZERO,
                          ecbptr()->celdbi);

if (newCache(&cache_name;
,
            &contrl_ptr->icontrol_dcacheToken,
            primeKeyLgh,
            secondaryKeyLgh,
            dataLgh,
            numbEntries,
            castOutTime,
            &cacheType,
            NULL) != CACHE_SUCCESS)
{
    printf("Error creating directory cache");
}
```

### Related Information

- “deleteCache—Delete a Logical Record Cache” on page 81
- “deleteCacheEntry—Delete a Cache Entry” on page 82
- “flushCache—Flush the Cache Contents” on page 198
- “readCacheEntry—Read a Cache Entry” on page 413
- “updateCacheEntry—Add a New or Update an Existing Cache Entry” on page 673.



---

## numbc—Query Number of Storage Blocks Available

This function returns the number of blocks currently available for a specified physical block type.

### Format

```
#include <sysapi.h>
int      numbc(enum p_blktype block_type);
```

#### **block\_type**

One of possible values representing a valid physical block type: LIOCB, L3, LECB, LSWB, LCOMMON, LFRAME or LMAX.

### Normal Return

Integer value representing the number of available storage blocks for this block type.

### Error Return

Not applicable.

### Programming Considerations

Authorization to issue a restricted macro is required.

### Examples

The following example obtains the number of available IOB blocks.

```
#include <sysapi.h>
int  iobs;
:
iobs = numbc(LIOCB);
```

### Related Information

None.

## open—Open a File

This function opens a file and returns an integer called a *file descriptor*.

You can use this file descriptor to refer to the file in subsequent input/output (I/O) operations; for example, in read or write functions. Each file opened by a process gets the lowest file descriptor not currently open for the process.

### Format

```
#include <fcntl.h>
int open(const char *pathname, int options, ...);
```

#### pathname

The **pathname** argument is a string giving the name of the file you want to open.

#### options

The integer **options** specifies options for the open operation by taking the bitwise inclusive OR of symbols defined in the `fcntl.h` header file. The options indicate whether the file should be accessed for reading, writing, reading and writing, and so on. Most open operations position a *file offset* (an indicator showing where the next read or write will take place in the file) at the beginning of the file; however, there are options that can change this position.

One of the following open options **must** be specified in the **options** argument.

- O\_RDONLY**    Open for reading only.
- O\_WRONLY**    Open for writing only.
- O\_RDWR**      Open for both reading and writing.

One of the following can also be specified in the **options** parameter:

**O\_APPEND**    Positions the file offset at the end of the file before each write operation.

**O\_CREAT**      Indicates that the call to open has a **mode** argument.

If the file being opened already exists, **O\_CREAT** has no effect except when **O\_EXCL** is also specified; see the **O\_EXCL** option, which follows.

The user ID of the file is set to the user ID of the process. If the parent directory of the file has its mode bit **S\_OSGID** set OFF, the group ID of the file is set to the group ID of the process; otherwise, the group ID of the file is set to the group ID of its parent directory.

If **O\_CREAT** is specified and the file did not previously exist, a successful open sets the modification time for the file. It also updates the modification time field in the parent directory.

#### TPF deviation from POSIX

The TPF system does not support the `atime` (access time) or `ctime` (change time) time stamp.

**O\_EXCL**        If both **O\_EXCL** and **O\_CREAT** are specified, open fails if the file already exists. If both **O\_EXCL** and **O\_CREAT** are specified

and **pathname** names a symbolic link, open fails regardless of the contents of the symbolic link.

## O\_NONBLOCK

When you are opening a character special file that supports a nonblocking open, O\_NONBLOCK controls whether subsequent reads and writes can block.

When you are opening a FIFO special file for reading only, if O\_NONBLOCK is not set (which is the default), the open function blocks the calling process until a process opens the file for writing. If O\_NONBLOCK is set the open function returns without delay.

When you are opening a FIFO special file for writing only, if O\_NONBLOCK is not set (which is the default), the open function blocks the calling process until a process opens the file for reading. If O\_NONBLOCK is set the open function returns an error if no process currently has the file open for reading.

## O\_TRUNC

Truncates the file to zero length under these conditions: (a) the file exists (b) the file is a regular file (c) the file is otherwise successfully opened with O\_RDWR or O\_WRONLY.

### TPF deviation from POSIX

The TPF system does not support the ctime (change time) time stamp. The mtime (modification time) time stamp is updated only when a file that is opened with write access, or read and write access is closed.

If O\_TRUNC is specified and the file previously existed, a successful open updates the modification time for the file.

.... An additional argument (...) is required if the O\_CREAT option is specified in **options**. This argument may be called the **mode** and has the mode\_t type. It specifies file permission bits to be used when a file is created. All the file permission bits are set to the bits of **mode** except for those set in the file-mode creation mask of the process. The following is a list of symbols that can be used for a mode.

## S\_ISUID

Privilege to set the user ID (UID) to run. When this file is run through the tpf\_fork function, the effective user ID of the process is set to the owner of the file. The process then has the same authority as the file owner rather than the authority of the actual caller.

## S\_ISGID

Privilege to set the group ID (GID) to run. When this file is run through the tpf\_fork function, the effective group ID of the process is set to the group of the file. The process then has the same authority as the file group rather than the authority of the actual caller.

## S\_IRUSR

Read permission for the file owner.

## S\_IWUSR

Write permission for the file owner.

## S\_IXUSR

Search permission (for a directory) for the file owner.

## S\_IRWXU

Read, write, and search for the file owner; S\_IRWXU is the bitwise inclusive OR of S\_IRUSR, S\_IWUSR, and S\_IXUSR.

## open

<b>S_IRGRP</b>	Read permission for the file group.
<b>S_IWGRP</b>	Write permission for the file group.
<b>S_IXGRP</b>	Search permission (for a directory) for the file group.
<b>S_IRWXG</b>	Read, write, and search permission for the file group. S_IRWXG is the bitwise inclusive OR of S_IRGRP, S_IWGRP, and S_IXGRP.
<b>S_IROTH</b>	Read permission for users other than the file owner.
<b>S_IWOTH</b>	Write permission for users other than the file owner.
<b>S_IXOTH</b>	Search permission for a directory for users other than the file owner.
<b>S_IRWXO</b>	Read, write, and search permission for users other than the file owner. S_IRWXO is the bitwise inclusive OR of S_IROTH, S_IWOTH, and S_IXOTH.

## Normal Return

If successful, open returns a file descriptor greater than or equal to zero.

## Error Return

If unsuccessful, open returns a -1 value and sets errno to one of the following:

<b>EACCES</b>	Access is denied. Possible reasons include: <ul style="list-style-type: none"><li>• The process does not have search permission on a component in <b>pathname</b>.</li><li>• The file exists, but the process does not have permission to open the file in the way specified by the flags.</li><li>• The file does not exist and the process does not have write permission on the directory where the file is to be created.</li><li>• O_TRUNC was specified, but the process does not have write permission on the file.</li></ul>
<b>EEXIST</b>	O_CREAT and O_EXCL were specified and either the named file refers to a symbolic link or the named file already exists.
<b>EINVAL</b>	The <b>options</b> parameter does not specify a valid combination of the O_RDONLY, O_WRONLY, and O_TRUNC bits.
<b>EISDIR</b>	<b>pathname</b> is a directory.
<b>ELOOP</b>	A loop exists in symbolic links. This error is issued if the number of symbolic links found while resolving the <b>pathname</b> argument is greater than POSIX_SYMLINK_MAX.
<b>EMFILE</b>	The process has reached the maximum number of file descriptors it can have open.
<b>ENAMETOOLONG</b>	<b>pathname</b> is longer than PATH_MAX characters, or some component of <b>pathname</b> is longer than NAME_MAX characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX.

<b>ENFILE</b>	The system has reached the maximum number of file descriptors that it can have open.
<b>ENOENT</b>	Typical causes: <ul style="list-style-type: none"> <li>• <code>O_CREAT</code> is not specified and the named file does not exist.</li> <li>• <code>O_CREAT</code> is specified and either the prefix of <b>pathname</b> does not exist or the <b>pathname</b> argument is an empty string.</li> </ul>
<b>ENOSPC</b>	The directory or file system intended to hold a new file does not have enough space.
<b>ENOTDIR</b>	A component of <b>pathname</b> is not a directory.
<b>ETPFNPIPWSYS</b>	In a loosely coupled environment, there was an attempt to access a FIFO special file (or named pipe) from a processor other than the processor on which the file was created.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

When a new regular file is created, if the `TPF_REGFILE_RECORD_ID` environment variable is set to a 2-character string, its value is used as the record ID for all pool file records that are allocated to store the contents of the file.

## Examples

The following example opens an output file for appending.

```
int fd;
fd = open("outfile", O_WRONLY | O_APPEND);
```

The following statement creates a new file with read, write, and execute permissions for the creating user. If the file already exists, open fails.

```
fd = open("newfile", O_WRONLY | O_CREAT | O_EXCL, S_IRWXU);
```

## Related Information

- “close—Close a File” on page 44
- “creat—Create a New File or Rewrite an Existing File” on page 54
- “dup—Duplicate an Open File Descriptor” on page 97
- “fcntl—Control Open File Descriptors” on page 129
- “fsync—Write Changes to Direct Access Storage” on page 232
- “lseek—Change the Offset of a File” on page 315
- “mkfifo—Make a FIFO Special File” on page 328
- “read—Read from a File” on page 410
- “umask—Set the File Mode Creation Mask” on page 659
- “write—Write Data to a File Descriptor” on page 696.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## opendir—Open a Directory

This function opens a directory so that it can be read with the `readdir` function.

### Format

```
#include <dirent.h>
DIR *opendir(const char *dirname);
```

**dirname**

The name of the directory to be opened.

The first `readdir` function call reads the first entry in the directory. The second `readdir` function call reads the second entry in the directory, and so on. You can only read a directory sequentially.

### Normal Return

If successful, the `opendir` function returns a pointer to a `DIR` object. This object describes the directory and is used in subsequent operations on the directory in the same way that `FILE` objects are used in file input/output (I/O) operations.

### Error Return

If unsuccessful, the `opendir` function returns a `NULL` pointer and sets `errno` to one of the following:

- |                     |  |
|---------------------|--|
| <b>EACCES</b>       | The process does not have permission to search some component of <b>dirname</b> or it does not have read permission on the directory itself.   |
| <b>ELOOP</b>        | A loop exists in the symbolic links. This error is issued if the number of symbolic links found while resolving the <b>dirname</b> argument is greater than <code>POSIX_SYMLINK_MAX</code> .   |
| <b>EMFILE</b>       | The process has too many other file descriptors open.  |
| <b>ENAMETOOLONG</b> | <b>pathname</b> is longer than <code>PATH_MAX</code> characters or some component of <b>pathname</b> is longer than <code>NAME_MAX</code> characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds <code>PATH_MAX</code> . |
| <b>ENFILE</b>       | The entire system has too many other file descriptors open.  |
| <b>ENOENT</b>       | The <b>dirname</b> directory does not exist.   |
| <b>ENOTDIR</b>      | Some component of the <b>dirname</b> path name is not a directory.   |

### Programming Considerations

None.

### Examples

The following example opens a directory.

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

void traverse(char *fn, int indent) {
    DIR *dir;
```

```

struct dirent *entry;
int count;
char path[1025];
struct stat info;

for (count=0; count<indent; count++) printf(" ");
printf("%s\n", fn);

if ((dir = opendir(fn)) == NULL)
    perror("opendir() error");
else {
    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_name[0] != '.') {
            strcpy(path, fn);
            strcat(path, "/");
            strcat(path, entry->d_name);
            if (stat(path, &info) != 0)
                fprintf(stderr, "stat() error on %s: %s\n", path,
                        strerror(errno));
            else if (S_ISDIR(info.st_mode))
                traverse(path, indent+1);
        }
    }
    closedir(dir);
}

main() {
    puts("Directory structure:");
    traverse("/dev", 0);
}

```

### Output

```

Directory structure:
/dev
/dev/null
/dev/tpf.omsg
/dev/tpf.imsf
/dev/tpf.socket.file

```

## Related Information

- “closedir—Close a Directory” on page 46
- “readdir—Read an Entry from a Directory” on page 415
- “rewinddir—Reposition a Directory Stream to the Beginning” on page 433.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## openlog–Open the System Control Log

This function opens a connection to the logging facility and sets process attributes that affect subsequent calls to the `syslog` function. Use this function with the `syslog` function and the `closelog` function to use the logging facilities provided with the `syslog` daemon.

See *TPF Transmission Control Protocol/Internet Protocol* for more information about the `syslog` daemon.

### Format

```
#include <syslog.h>
void      openlog(const char *ident, int logopt, int facility);
```

#### **ident**

A pointer to a string that is prefixed to every message or a NULL pointer.

#### **logopt**

Specify the following value or NULL:

##### **LOG\_PID**

Log the process ID with each message. This is useful for identifying specific processes.

#### **facility**

Specifies a default facility to be assigned to all messages that do not have an explicit facility already coded. If you specify NULL, the initial default facility is LOG\_USER. See “syslog–Send a Message to the Control Log” on page 523 for more information about the other values that can be specified.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- It is not necessary to call the `openlog` function before calling the `syslog` function. If you do not call the `openlog` function, the `openlog` function is called automatically the first time the `syslog` function is called. If you do not call the `openlog` function, messages will be logged without a prefix and any message without an explicit facility will be logged to the destination defined for the LOG\_USER facility.
- This function is implemented in dynamic link library (DLL) CTXO. You must use the definition side-deck for DLL CTXO to link-edit an application that uses this function.

### Examples

See “syslog–Send a Message to the Control Log” on page 523 for an example using the `openlog` function.

### Related Information

- “closelog–Close the System Control Log” on page 48
- “syslog–Send a Message to the Control Log” on page 523.



## pausc—Control System Multiprocessor Environment

This function permits an E-type program to request the control program establish a uniprocessor (UP) environment or reestablish the multiprocessor (MP) environment in a system running on a multiple I-stream central processing complex (CPC). It permits a program that is not capable of executing in an MP environment to complete its processing in a UP environment.

### Format

```
#include <sysapi.h>
void      pausc(enum t_pausc_opt option);
```

#### option

Either PAUSC\_BEGIN or PAUSC\_END to turn on or turn off the multiprocessor environment. The definition of t\_pausc\_opt is defined in the sysapi.h header file.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- This function can only be run on the main I-stream.
- Use this function with discretion. Executing in a UP environment can seriously impact performance on an MP CPC. The time spent in the UP environment must be kept to a minimum. The number of invocations of this function must also be kept to a minimum.
- The same ECB must stop and restart the MP environment.
- No memory locks or record holds are permitted while the system is in a UP environment (paused).
- When requesting a system UP pause, this ECB must not have previously paused the system. If it has, system error dump CTL-000578 is taken and control is returned to the program.
- When requesting a system MP restart, this ECB must have previously paused the system. If not, the entry is exited and a 0005777 system error is issued.

### Examples

```
#include <sysapi.h>
. /* processing in MP mode, main I-stream */
pausc(PAUSC_BEGIN);
. /* processing in UP mode */
pausc(PAUSC_END)
. /* return to MP mode */
```

### Related Information

None.

## pause—Wait for a Signal

This function suspends the calling process until a signal is delivered.

### Format

```
#include <signal.h>
int pause(void);
```

This function does not have any parameters.

### Normal Return

The pause function suspends the calling process until a signal causes control to be returned. When a signal is received, if the corresponding signal handler returns control to the calling process, the pause function returns a value of `-1` and sets `errno` to **EINTR**, which indicates that pause function processing was interrupted by a signal.

### Error Return

None.

## Programming Considerations

The behavior of the signal handler associated with a signal that occurs while the calling process is suspended can cause the entry control block (ECB) to exit.

## Examples

The following example suspends processing until any signal arrives.

```
#include <signal.h>
...
{
    /* wait for any signal */
    pause();

    /* we only expect to get control here if pause was */
    /* interrupted by a signal for which a signal handler */
    /* was called and only if that signal handler returned */
    ...
}
```

## Related Information

- “sigaction—Examine and Change Signal Action” on page 486
- “signal—Install Signal Handler” on page 495
- “sigpending—Examine Pending Signals” on page 498
- “sigprocmask—Examine and Change Blocked Signals” on page 499
- “sigsuspend—Set Signal Mask and Wait for a Signal” on page 502.

## perror—Write Error Message to Standard Error Stream

This function prints an error message to stderr.

### Format

```
#include <stdio.h>
void perror(const char *string);
```

#### string

The text prefixed to the error message written to stderr.

If **string** is not NULL and it 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. The message associated with the value in `errno` is then printed, followed by a new-line character. The content of the message is the same as the content of a string returned by the `strerror` function when passed the value of `errno`.

To produce accurate results, ensure that the `perror` function is called immediately after a library function returns with an error; otherwise, subsequent calls may change the `errno` value.

If the error is associated with the `stderr` file, a call to `perror` is not valid.

#### TARGET(TPF) restriction

Avoid the TARGET(TPF) implementation of the `perror` function because it is nonstandard. It requires the application to set `errno` to a system error number and cause a system error with return using the **string** parameter as the appended console message. Instead of `perror` in TARGET(TPF), use the `snpc` function or one of the `serrc_op` family of functions.

### Normal Return

Returns no value.

### Error Return

Not applicable.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

## Examples

The following example tries to open a stream. If the `fopen` function fails, the example prints a message and ends the program.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fh;

    if ((fh = fopen("myfile.dat", "r")) == NULL) {
```

## **perror**

```
        perror("Could not open data file");  
        abort();  
    }  
}
```

## **Related Information**

“strerror—Get Pointer to Run-Time Error Message” on page 517.

## pipe—Create an Unnamed Pipe

This function creates a *pipe*, which is an input/output (I/O) channel that one or more processes can use to communicate with each other or, in some cases, that a process can use to communicate with itself.

Data is written into one end of the pipe and read from the other.

### Format

```
#include <unistd.h>
int pipe(int fdinfo[2]);
```

#### **fdinfo**

The **fdinfo** parameter points to an array of 2 integers where the pipe function stores two file descriptors.

### Normal Return

If successful, the pipe function returns a zero value. A file descriptor for the read end of the pipe is stored in the **fdinfo[0]** parameter and a file descriptor for the write end of the pipe is stored in the **fdinfo[1]** parameter.

### Error Return

If unsuccessful, the pipe function returns a value of `-1` and sets `errno` to one of the following:

#### **EMFILE**

The process has already reached its maximum number of open file descriptors. This limit is given by `OPEN_MAX`, which is defined in the `limits.h` header file.

#### **ENOMEM**

There is not enough storage space available.

## Programming Considerations

- Processes can read from the file descriptor specified by the **fdinfo[0]** parameter (also referred to as the read end of the pipe) and write to the file descriptor specified by the **fdinfo[1]** parameter (also referred to as the write end of the pipe). Data written to the file descriptor specified by the **fdinfo[1]** parameter is read from the file descriptor specified by the **fdinfo[0]** parameter on a first-in-first-out (FIFO) basis.
- When a pipe is created, the `O_NONBLOCK` and `FD_CLOEXEC` flags are turned off for both ends of the pipe. These flags can be set on with the control open file descriptors (`fcntl`) function. See “`fcntl`—Control Open File Descriptors” on page 129 for more information.
- If a process reads from the read end of the pipe, the pipe is empty, and the `O_NONBLOCK` flag is off, the process is blocked until another process writes to the write end of the pipe. Similarly, if a process writes to the write end of the pipe, the pipe becomes full, and the `O_NONBLOCK` flag is off, the process is blocked until another process reads enough data from the read end of the pipe to make space in the pipe for the write to be completed successfully.
- All types of seek operations (such as `fseek` or `fgetpos`) fail on pipes.
- When the write end of the pipe is closed, processes can continue to read from the read end until the pipe becomes empty. After the pipe is empty, any additional reads return an end-of-file condition.

## pipe

- If the read end of the pipe is closed while the write end remains open, any additional writes return an error indicator, `errno` is set to `EPIPE`, and a `SIGPIPE` signal is generated.
- If the `pipe` function successfully creates a pipe, the access, change, and modification times for the pipe are updated in the associated file descriptors.

## Examples

The following example consists of two programs that calculate and write the first 20 numbers in the Fibonacci sequence, which is a sequence of numbers where each number is the sum of the 2 preceding numbers.

Program 1 is a simple program that reads 2 integers, adds the integers, and writes the sum.

Program 2 opens two pipes that are used to communicate with program 1, which is created as a new child process using the `tpf_fork` function. For the child process, the read end of one pipe is set as standard input (`stdin`) and the write end of the other pipe is set as standard output (`stdout`).

The two processes communicate by using the two pipes to calculate the first 20 numbers in the Fibonacci sequence.

The following sequence is the result of the processing of these two programs:

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

Program 1:

```
/******  
/* A simple integer adder.                               */  
/******  
#include <stdio.h>  
  
int main(void)  
{  
    int i1, i2;  
  
    /******  
    /* Read two numbers from stdin.                       */  
    /******  
    while (scanf("%d%d", &i1, &i2) == 2)  
    {  
        /******  
        /* Write their sum to stdout.                     */  
        /******  
        printf("%d\n", i1 + i2);  
        fflush(stdout);  
    }  
    return 0;  
}
```

Program 2:

```
/******  
/* This program calls the simple integer adder program to calculate */  
/* the first 20 numbers in the fibonacci series.                   */  
/******  
#define _POSIX_SOURCE  
#include <stdio.h>  
#include <unistd.h>  
#include <fcntl.h>  
#include <sysapi.h>
```

```

#define SERIES_COUNT 20

int main(int argc, char **argv)
{
    FILE *original_stdout;
    int pipe1[2], pipe2[2];
    FILE *write_to_adder, *read_from_adder;

    {
        /******
        /* Copy the original stdout. We'll use its file descriptor */
        /* for the child process' stdout. */
        /******
        int stdout_copy = dup(STDOUT_FILENO); /* file descriptor 3 */
        original_stdout = fdopen(stdout_copy, "w");
        fcntl(stdout_copy, F_SETFD,
            FD_CLOEXEC | fcntl(stdout_copy, F_GETFD));
    }

    /******
    /* Close file descriptors 0 and 1. We will use them for the */
    /* child process' stdin and stdout. */
    /******
    fclose(stdin); /* file descriptor 0 */
    fclose(stdout); /* file descriptor 1 */

    /******
    /* The first pipe() function call opens file descriptors 0 and 1. */
    /* We will duplicate and close 0 for the parent to read from the */
    /* child process; 1 will be the child process' stdout. */
    /******
    pipe(pipe1); /* file descriptors 0 and 1 */
    {
        int read_end_copy = dup(pipe1[0]); /* file descriptor 4 */
        close(pipe1[0]); /* file descriptor 0 */
        read_from_adder = fdopen(read_end_copy, "r");
        fcntl(read_end_copy, F_SETFD,
            FD_CLOEXEC | fcntl(read_end_copy, F_GETFD));
    }

    /******
    /* The second pipe() function call opens file descriptors 0 and */
    /* 5. We will use 5 to write to the child process; 0 will be the */
    /* child process' stdin. */
    /******
    pipe(pipe2); /* file descriptors 0 and 5 */
    write_to_adder = fdopen(pipe2[1], "w");
    fcntl(pipe2[1], F_SETFD, FD_CLOEXEC | fcntl(pipe2[1], F_GETFD));

    /******
    /* The file descriptors are now set as follows: */
    /* */
    /* 0 (pipe2[0])---child's stdin */
    /* 1 (pipe1[1])---child's stdout */
    /* 2 (stderr) */
    /* 3 (original_stdout) */
    /* 4 (read_from_adder) */
    /* 5 (write_to_adder) */
    /* */
    /* Start the simple integer adder as a child process. Note: the */
    /* tpf_fork() function requires restricted macro permission. */
    /******
    {
        struct tpf_fork_input tfi = {
            "bin/simple_integer_adder",
            TPF_FORK_FILE,
            TPF_FORK_IS_BALANCE

```

## pipe

```
    };
    tpf_fork(&tfi);
}

/*****
 * Close the ends of the pipes that the child is using.
 */
close(pipe2[0]);
close(pipe1[1]);

/*****
 * Calculate the fibonacci series.
 */
{
    int counter = SERIES_COUNT;
    int last = 1;
    int next = 1;

    fprintf(original_stdout, "%d", last);

    while (--counter)
    {
        fprintf(write_to_adder, "%d %d\n", last, next);
        fflush(write_to_adder);
        last = next;
        fprintf(original_stdout, " %d", last);
        fscanf(read_from_adder, "%d", &next);
    }
}

/*****
 * Close the working file streams.
 */
fclose(original_stdout);
fclose(write_to_adder);
fclose(read_from_adder);

return 0;
}
```

## Related Information

- “close—Close a File” on page 44
- “fcntl—Control Open File Descriptors” on page 129
- “mkfifo—Make a FIFO Special File” on page 328
- “open—Open a File” on page 380
- “read—Read from a File” on page 410
- “write—Write Data to a File Descriptor” on page 696.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



## postc—Mark Event Element Completion

This function defines to the control program the completion and success of an element of a named event. This function is used with the `evntc` and `evnwc` functions.

### Format

```
#include <tpfapi.h>
int      postc(struct ev0bk  *evninf,
               enum t_evn_typ type,
               int           ercode);
```

#### evninf

A pointer to the `postc` parameter block. See `struct ev0bk` and `struct tpf_ev0bk_list_data` for more information.

#### type

The type of event element being completed. The argument must belong to the enumerated type `t_evn_typ`, defined in `tpfapi.h`. Use the following predefined terms:

#### EVENT\_MSK

for mask events

#### EVENT\_CNT

for count events

#### EVENT\_CB\_Dx

where *x* is a single hexadecimal digit (**0–F**) for core events

#### EVENT\_LIST

for list events.

#### ercode

Indicates if the post is an error post and causes the event completion to be signaled as completed with error. A value of zero indicates that the post was not an error post. A nonzero value contains the error code. The error code can range from 1–256.

### Normal Return

Integer value of zero. If **EVENT\_CB\_Dx** or **EVENT\_LIST** is coded for `type`, the specified core level is marked as not holding a block.

### Error Return

One of the following:

- A system error with exit results if the type of post and the type of the event do not match.
- An integer value of 1 is returned if the event name does not exist.
- An integer value of 2 is returned if a list event data item is not found.

### Programming Considerations

- For count events, `evninf->evnbkm2` can contain a 16-bit mask that is ORed with the POST MASK 2 field of the event. The current value of the event is automatically decremented by 1 when the post request is issued.
- For mask events, `evninf->evnpstinf.evnbkm1` must contain a 16-bit mask to be converted to a 1's complement and ANDed against the mask value in the named event. `evninf->evnbkm2` can contain a 16-bit mask to be ORed with the POST MASK 2 field in the event.

## postc

- For list events, the list contains one or more data items. All data items in the list are posted to the list event.

## Examples

The following example shows 2 ECB-controlled tasks. The first defines a count event and waits for its completion. The second task at some later time posts the completion of the event.

### Task 1

```
struct ev0bk eb =
    { { "my_event" }, 1 };
evntc(&eb, EVENT_CNT, 'Y',
      250, EVNTC_NORM);
evnwc(&eb, EVENT_CNT);
/* this task is suspended */
/* until "my_event" completes */

/* after "my_event" completes */
/* Task 1 resumes execution */
```

### Task 2

```
struct ev0bk pb =
    { { "my_event" } };
postc(&pb, EVENT_CNT, 0);
/* post "my_event" with no */
/* error */
```

## Related Information

- “evinc—Increment Count for Event” on page 108
- “evnqc—Query Event Status” on page 109
- “evntc—Define an Internal Event” on page 111
- “evnwc—Wait for Event Completion” on page 113
- “tpf\_genlc—Generate a Data List” on page 599
- “tpf\_sawnc—Wait for Event Completion with Signal Awareness” on page 627.

---

**printf–Format and Write Data**

The information for this function is included in “fprintf, printf, sprintf–Format and Write Data” on page 201.

## progc—Return Program Allocation Table (PAT) Slot Address

This function accepts a program name and returns the program's base PAT slot address. A program's PAT slot contains that program's characteristics (residency, mode, macro authorization, and so on), and also that program's current core and file addresses.

### Format

```
#include <c$idspat.h>
struct pat *progc(const char *name, enum t_progc mdbf_ind);
```

#### name

This argument is a pointer to a 4-character program name whose PAT slot address will be returned.

#### mdbf\_ind

This argument specifies which multiple database function (MDBF) index field is used when retrieving the PAT entry as well as which PAT slot to retrieve (the parent slot or the transfer vector (TV) slot). This argument must belong to the enumerated type `t_progc`, defined in the `tpfapi.h` header file.

### Normal Return

The base PAT slot address of the specified program is returned.

If the program name supplied is a transfer vector, the parent base PAT slot address is returned, unless the TV slot is requested.

If the program name supplied is I-stream unique, the program's base PAT slot address for the I-stream that called the macro is returned.

### Error Return

NULL is returned if the specified program name could not be found in the PAT.

## Programming Considerations

During normal system operation, only 1 PAT slot will exist for each program on a given subsystem; however, if the E-type loader is used, multiple PAT slots can exist for a given program. The PAT slot address returned is always the address of the base allocated program's PAT slot.

**Note:** The `mdbf_ind` field, if coded, must specify one of the following:

- `PROGC_PBI` for the ECB program base ID (CE1PBI)
- `PROGC_DBI` for the database ID (CE1DBI)
- `PAT_PBI` for the TV with program base ID
- `PAT_DBI` for the TV with database ID.

## Examples

- The following example retrieves the PAT slot address for program ABCD and stores it in field `PAT_SLOT`.

```
#include <c$idspat.h>
...
struct pat *pat_slot;
pat_slot = progC( "ABCD", PROGC_PBI );
```

- The following example retrieves the PAT slot address for program CVII and stores it in field `PAT_SLOT`. Program CVII is a TV in CVIA.

```
#include <c$idspat.h>
:
struct pat *pat_slot;
pat_slot = progC( "CVII", PAT_PBI );
```

## **Related Information**

None.

## putc, putchar—Write a Character

This function writes a character to a stream.

### Format

```
#include <stdio.h>
int putc(int c, FILE *stream);
int putchar(int c);
```

**c** The character to be written.

#### **stream**

The stream on which the character is to be written.

This function converts **c** to unsigned char and then writes **c** to the output **stream** at the current position. The putchar function is identical to:

```
putc(c, stdout);
```

These functions are also available as macros. For better performance use the macro forms rather than the functional forms.

By default, if the `stdio.h` header file is included, the macro is called. Therefore, the arguments should not be expressions that cause the values of variables to change.

You can access the actual function by using one of the following methods:

- Specify `#undef`; for example, `#undef putc`
- Surround the function name with parentheses; for example: `(putchar)('a')`

The `putc` and `putchar` functions have the same restriction as any write operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an end-of-file (EOF) has been reached.

### Normal Return

The `putc` and `putchar` functions return the character that is written.

### Error Return

A return value of EOF indicates an error.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

### Examples

The following example writes the contents of a buffer to a data stream. The body of the `for` statement is null because the example carries out the writing operation in the test expression.

```
#include <stdio.h>
#define LENGTH 80

int main(void)
{
    FILE *stream = stdout;
```

```
int i, ch;
char buffer[] = "Hello world";

/* This could be replaced by using the fwrite routine */
for ( i = 0;
      (i < strlen(buffer)) && ((ch = putc(buffer[i], stream)
      ++i);
}
```

**Output**

Hello world

**Related Information**

- “fputc—Write a Character” on page 209
- “fwrite—Write Items” on page 239
- “getc, getchar—Read a Character from Input Stream” on page 246.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## puts—Put String to Standard Output Stream

This function writes a string to the standard output stream.

### Format

```
#include <stdio.h>
int puts(const char *string);
```

#### **string**

The text to be written to stdout.

This function writes the string pointed to by **string** to the stream pointed to by stdout and appends the new-line character to the output. The terminating null character is not written.

### Normal Return

Returns the number of bytes written.

### Error Return

If an error occurs, puts returns EOF. If a system write error occurs, the write stops at the point of failure.

## Programming Considerations

The puts function has the same restriction as any write operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an end-of-file (EOF) has been reached.

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

#### **TARGET(TPF) restrictions**

- The puts function in the TARGET(TPF) library is a limited and nonstandard substitute for the standard puts function. The TARGET(TPF) library has no support for files, stdout, or file redirection.
- The TARGET(TPF) version of the puts function does not have the same effect as the ISO-C version. Use of both versions in the same application can have results that cannot be predicted.
- Owing to the large diversity of terminal devices and network support in the TPF system, the installation must code the TARGET(TPF) version of this function to work with its requirements. All programming considerations related to the use of this function, therefore, depend on the manner of implementation.
- The TARGET(TPF) version of this function requires EBROUT to be set to a valid address. If the entry control block (ECB) that uses this function was created via the entry creation functions (such as credc), the application is responsible for passing the desired value of EBROUT to the ECB that was created.



## Examples

The following example writes 'Hello World' to stdout.

```
#include <stdio.h>

int main(void)
{
    if ( puts("Hello World") == EOF )
        printf( "Error in puts\n" );
}
```

### Output

Hello World

## Related Information

- “fputs—Write a String” on page 211
- “gets—Obtain Input String” on page 273.

## raisa—General File Get File Address

This function is used to generate the address of a subsequent record on a general file when the current position is known.

### Format

```
#include <tpfapi.h>
void      raise(enum t_blktype size, enum t_lvl level, int count,
               int type, int dev);
```

#### size

The record size to which the target general file has been formatted. This argument must belong to enumeration type `t_blktype`, which is defined in `tpfapi.h`. Use Lx notation, as follows:

- L1** 381 bytes
- L2** 1055 bytes
- L4** 4095 bytes.

#### level

One of 16 possible values representing a valid data level from enumeration type `t_lvl`, expressed as Dx, where x represents the hexadecimal number of the level (0–F). The FARW on this level contains the file address of the current file position.

#### count

This argument is an unsigned integer representing the number of records to add to the current file address to arrive at the requested position or file address. Do **not** use a negative value as this argument.

#### type

The type of addressing scheme used to address the records on the file. Use the defined terms **RAISA\_RRN** to reflect relative record number addressing, or **RAISA\_MCHR** to reflect absolute module/cylinder/head/record addressing.

#### dev

The device type on which the general file resides. The supported device types are: 3390, 3380, 3375, and 3350. Use the defined terms **RAISA\_D3390** to indicate 3390, **RAISA\_D3380** to indicate 3380, **RAISA\_D3375** to indicate 3375, or **RAISA\_D3350** to indicate 3350.

### Normal Return

Void. The FARW has been updated with the proper file address.

### Error Return

Not applicable.

### Programming Considerations

- This function is provided for RRN and 4-byte MCHR formats for online programs only.
- Specifying an invalid parameter results in a system error with exit.

### Examples

The following example increments the RRN file address on level D3 by one 1055-byte record for a general file on a 3380.

```
#include <tpfapi.h>
...
raisa(L2,D3,1,RAISA_RRN,RAISA_D3380);
```

## Related Information

- “gdsnc—Get Data Set Entry” on page 241
- “gdsrsc—Get General Data Set Record” on page 244.

---

## raise—Raise Condition

This function sends the signal **sig** to the program that called the `raise` function.

Use the `sigaction` function or the `signal` function to specify how a signal is handled when the `raise` function is called.

Default processing for signals when the signal handler is `SIG_DFL` is in internal library function `__sigdfp`. See Table 13 on page 495 for `SIG_DFL` handler action for the signals. The `SIG_DFL` handler actions can be changed by modifying `__sigdfp` code in `CSIGDP`.

### Format

```
#include <signal.h>
int raise(int sig);
```

#### **sig**

The signal sent to the program that issued the `raise` function.

See Table 13 on page 495 for the list of supported signals.

### Normal Return

The returned value is 0.

### Error Return

A nonzero error return indicates that the function was not successful.

## Programming Considerations

- There is no support for static storage base switching when activating the signal processing function. If the signal handler function is in the dynamic load module (DLM) where the `signal` or `sigaction` function is called to set the handler, the `raise` function for that signal can only be called from the same DLM.
- The `raise` function sets the handler for the signal to `SIG_DFL` before the current handler is called.
- After the `raise` function sends the specified signal to the calling process, it attempts to handle any unhandled signals (including the signal that was just sent) before returning to the caller. This means that a call to the `raise` function can result in signals (other than the signal that was raised) being handled. Note that blocked signals, including the signal that was raised, cannot be handled immediately.
- The `raise` function sets the signal handler for the signal (specified by the **sig** parameter) to `SIG_DFL` (in the `sa_handler` field in the `sigaction` data structure) before the signal handler is given control if either of the following conditions is true:
  - The signal handler was installed by the `signal` function.
  - The signal handler was installed by the `sigaction` function and the `SA_RESETHAND` flag was set in the `sa_flags` field in the `sigaction` data structure when the `sigaction` function was called.

If the signal handler was installed by the `sigaction` function and the `SA_RESETHAND` flag was not set in the `sa_flags` field in the `sigaction` data structure when the `sigaction` function was called, the signal handler remains installed.

## Examples

This example establishes a signal handler called `sig_hand` for the signal `SIGUSR1`. The signal handler is called whenever the `SIGUSR1` signal is raised; it ignores the first 9 occurrences of the signal. On the tenth raised signal, it prints a message and returns without establishing the signal handler again. Note that the signal handler must be established again each time it is called.

**Note:** The signal handler must be established each time the `raise` function is called because the signal handler is installed by the `signal` function.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
void sig_hand(int); /* declaration of sig_hand() as a function */
int i;
void ADLM(void)
{
    signal(SIGUSR1, sig_hand); /* set up handler for SIGUSR1 */
    for (i=0; i<10; ++i)
        raise(SIGUSR1); /* signal SIGUSR1 is raised */
} /* sig_hand() is called */
void sig_hand(int);
static int count = 0; /* initialized only once */
count++;
if (count == 10) { /* ignore the first 9 occurrences of this signal */
    printf("reached 10th signal\n");
    return;
}
else
    signal(SIGUSR1, sig_hand); /* set up the handler again */
}
```

## Related Information

- “abort—Terminate Program Abnormally” on page 7
- “sigaction—Examine and Change Signal Action” on page 486
- “signal—Install Signal Handler” on page 495
- “sigpending—Examine Pending Signals” on page 498
- “sigprocmask—Examine and Change Blocked Signals” on page 499
- “sigsuspend—Set Signal Mask and Wait for a Signal” on page 502.

## rcunc—Release Core Block and Unhold File Record

This macro returns a working storage block to the system and removes a record from the record hold table.

### Format

```
#include <tpfio.h>
void rcunc (enum t_lvl level);
```

or

```
#include <tpfio.h>
void rcunc (TPF_DECB *decb);
```

#### level

One of 16 possible values representing a valid entry control block (ECB) data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter identifies the data level containing the address of the working storage block to be returned and the address of the file record to be unheld.

#### decb

A pointer to a data event control block (DECB). This parameter identifies the DECB containing the address of the working storage block to be returned and the address of the file record to be unheld.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- The specified ECB data level or DECB must be occupied with a valid working storage block address and the record address on the specified ECB data level or DECB must have been placed in the record hold table (RHT) by the calling ECB.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.

### Examples

The following example releases a core block and unholds the record from data level 2.

```
#include <tpfapi.h>
#include <tpfio.h>
:
:
rcunc(D2);          /* Release block and unhold record */
```

The following example releases a core block and unholds the record from a DECB.

```
#include <tpfapi.h>
#include <tpfio.h>
:
:
TPF_DECB *decb;
:
:
rcunc(decb);        /* Release block and unhold record */
```

## Related Information

- “relcc—Release Working Storage Block” on page 421
- “unfrc—Unhold a File Record” on page 661
- “tpf\_rcrhc—Release a Core Block and File Address” on page 624.

See *TPF Application Programming* for more information about DECBs.

## read—Read from a File

This function reads from a file.

### Note

This description applies only to files. See *TPF Transmission Control Protocol/Internet Protocol* for more information about the read function for sockets.

### TPF deviation from POSIX

The TPF system does not support the `atime` (access time) time stamp.

## Format

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t N);
```

### fildes

The file descriptor.

### buf

The buffer or memory area.

**N** The number of bytes of data to be read.

From the file indicated by file descriptor **fildes**, the read function reads **N** bytes of input into the memory area indicated by **buf**.

If **fildes** refers to a regular file or any other type of file on which the process can seek, the read function begins reading at the file offset associated with **fildes**. If successful, the read function changes the file offset by the number of bytes read. **N** should not be greater than `INT_MAX` (defined in the `limits.h` header file).

If **fildes** refers to a file on which the process cannot seek, read begins reading at the current position. There is no file offset associated with such a file.

## Normal Return

If successful, read returns the number of bytes that are actually read and placed in **buf**, equal to the number of bytes requested (**N**).

The return value is less than **N** only if:

- The read function reached the end of the file before reading the requested number of bytes.
- The file is a nonblocking character special file that has fewer than **N** bytes immediately available for reading.

If the starting position for the read operation is at the end of the file or beyond, read returns a 0 value.

If the read function is reading a regular file and finds a part of the file that has not been written (but before the end of the file), read places 0 bytes into **buf** in place of the unwritten bytes.



If a read function is attempted on an empty FIFO special file and no process has opened the file for writing, the read function returns a value of 0 to indicate the end of the file. When the buffer for a FIFO special file becomes full, the writer is blocked until enough data is removed from the file with a read function to allow the write function to be completed.

If the number of bytes of input that you want to read is 0, the read function simply returns a value of 0 without attempting any other action.

## Error Return

If the read function fails, it returns a value of `-1` and sets `errno` to one of the following:

<b>EAGAIN</b>	<code>O_NONBLOCK</code> is set to 1 but data was not available for reading.
<b>EBADF</b>	<code>fd</code> is not a valid open file descriptor.
<b>EINTR</b>	A signal interrupted read function processing before any data was available.
<b>EINVAL</b>	<code>N</code> contains a value that is less than 0, or the request is not valid or not supported.

## Programming Considerations

None.

## Examples

The following example opens a file and reads input.

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

main() {
    int ret, fd;
    char buf[1024];

    if ((fd = open("my.data", O_RDONLY)) < 0)
        perror("open() error");
    else {
        while ((ret = read(fd, buf, sizeof(buf)-1)) > 0) {
            buf[ret] = 0x00;
            printf("block read: \n<%s>\n", buf);
        }
        close(fd);
    }
}
```

### Output

```
block read:
<ask dad;
sad lad ask dad fad daf lak;
jasf dasd fall slaj fask slak flak;
flask skald salsa slkja dsalk fjakl;
fadjak lakkad skalla aladja kfslsa;
>
```

## Related Information

- “close—Close a File” on page 44
- “creat—Create a New File or Rewrite an Existing File” on page 54

## read

- “dup—Duplicate an Open File Descriptor” on page 97
- “fcntl—Control Open File Descriptors” on page 129
- “fread—Read Items” on page 213
- “lseek—Change the Offset of a File” on page 315
- “open—Open a File” on page 380
- “write—Write Data to a File Descriptor” on page 696.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## readCacheEntry–Read a Cache Entry

This function reads a cache entry.

### Format

```
#include <c$cach.h>
long readCacheEntry( const cacheToken * cache_to_read),
                    const void *primary_key,
                    const long *primary_key_length,
                    const void *secondary_key,
                    const long *secondary_key_length,
                    long *size_of_buffer,
                    void *buffer);
```

#### **cache\_to\_read**

The returned cache\_token from the newCache function that defined the logical record cache.

#### **primary\_key**

A pointer to a field that contains the value of the primary key.

#### **primary\_key\_length**

The length of the specified primary key. The length must be equal to or less than the maximum value specified on the newCache function.

#### **secondary\_key**

A pointer to a field that contains the value of the secondary key. If a secondary key length was not specified on the newCache function, this field is ignored and will contain a null pointer.

#### **secondary\_key\_length**

The length of the specified secondary key. The length must be equal to or less than the maximum value specified on the newCache function.

#### **size\_of\_buffer**

A pointer to a field that contains the length of the passed buffer. This field is overlayed with the actual length of the data entry.

#### **buffer**

A pointer to the area that is to be overlayed with the contents of the specified cache entry if the entry is found; otherwise, the buffer remains unchanged.

### Normal Return

#### **CACHE\_SUCCESS**

The function is completed successfully.

### Error Return

One of the following:

#### **CACHE\_ERROR\_HANDLE**

The cache\_token provided for the cache\_to\_read parameter is not valid.

#### **CACHE\_ERROR\_PARAM**

The value passed for one of the parameters is not valid.

#### **CACHE\_NOT\_FOUND**

The cache entry was not read because it was not found in the cache.

#### **CACHE\_ERROR\_CACHE**

The cache address has changed.

## readCacheEntry

### Programming Considerations

The primary and secondary keys must exactly match the primary and secondary keys used to add an entry to the cache. The entry control block (ECB) must have the same database ID (DBI) as the ECB used to create the entry.

### Examples

The following example reads a cache entry from the file system directory cache.

```
#include <c$cach.h>
#include <i$glue.h>

struct itran *tran;
long  primaryKeyLgh = strlen( tran->itrans_response.itres_name );
long  secondaryKeyLgh = sizeof(ino_t);
struct TPF_directory_entry tde;
long  bufferSize= sizeof(struct TPF_directory_entry);
struct icontrl * contrl_ptr; /* pointer file system control area */

contrl_ptr = cinfc_fast_ss(CINFC_CMMZERO,
                          ecbptr()->celdbi);

if ((contrl_ptr->icontrl_dcachetoken.token1 == 0)
    ||
    (readCacheEntry(&contrl_ptr->icontrl_dcachetoken,
                  &tran->itrans_response.itres_name,
                  &primaryKeyLgh,
                  &tran->itrans_response.
                    itres_parent_inode.inode_ino,
                  &secondaryKeyLgh,
                  &bufferSize,
                  &tde) != CACHE_SUCCESS))
[
    /* read directory entry from file */
]
```

### Related Information

- “deleteCache—Delete a Logical Record Cache” on page 81
- “deleteCacheEntry—Delete a Cache Entry” on page 82
- “flushCache—Flush the Cache Contents” on page 198
- “newCache—Create a New Logical Record Cache” on page 376
- “updateCacheEntry—Add a New or Update an Existing Cache Entry” on page 673.

## readdir—Read an Entry from a Directory

This function reads an entry from a directory.

### Format

```
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

#### **dir**

A pointer to a **dir** object returned by the `opendir` function.

This function returns a pointer to a `dirent` structure describing the next directory entry in the directory stream associated with **dir**. A call to the `readdir` function overwrites data produced by a previous call to `readdir` on the same directory stream. Calls for different directory streams do not overwrite the data of each directory stream.

#### TPF deviation from POSIX

The TPF system does not support the `atime` (access time) time stamp.

A `dirent` structure contains character pointer **d\_name**, which points to a string that gives the name of a file in the directory. This string ends in a terminating null and has a maximum of `NAME_MAX` characters.

Save the data from `readdir`, if required, before calling `closedir` because the `closedir` function frees the data.

### Normal Return

If successful, the `readdir` function returns a pointer to a `dirent` structure describing the next directory entry in the directory stream. When the `readdir` function reaches the end of the directory stream, it returns a NULL pointer.

### Error Return

If unsuccessful, the `readdir` function returns a NULL pointer and sets `errno` to the following:

**EBADF**            **dir** does not yield an open directory stream.

### Programming Considerations

None.

### Examples

The following example reads the contents of a root directory.

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

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

    if ((dir = opendir("/")) == NULL)
        perror("opendir() error");
    else {
```

## readdir

```
puts("contents of root:");
while ((entry = readdir(dir)) != NULL)
    printf("  %s\n", entry->d_name);
closedir(dir);
}
```

### Output

```
contents of root:
.
..
dev
usr
```

## Related Information

- “closedir—Close a Directory” on page 46
- “opendir—Open a Directory” on page 384
- “rewinddir—Reposition a Directory Stream to the Beginning” on page 433.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## readlink—Read the Value of a Symbolic Link

This function reads the value of a symbolic link.

### Format

```
#include <unistd.h>
int readlink(const char *path, char *buf, size_t bufsiz);
```

#### **path**

The name of a symbolic link.

#### **buf**

The address of the array of characters where the symbolic link is to be read.

#### **bufsiz**

The number of the character to read.

This function places the contents of symbolic link **path** in buffer **buf**. The size of the buffer is indicated by **bufsiz**. The result stored in **buf** does not include a terminating null character.

If the buffer is too small to contain the value of the symbolic link, that value is truncated to the size of the buffer (**bufsiz**). If the value returned is the size of the buffer, use the `lstat` function to determine the actual size of the symbolic link.

### Normal Return

When **bufsiz** is greater than 0 and the `readlink` function has completed successfully, the `readlink` function returns the number of bytes placed in the buffer. When **bufsiz** is 0 and the `readlink` function has otherwise completed successfully, the `readlink` function returns the number of bytes contained in the symbolic link and the buffer is not changed.

If the returned value is equal to **bufsiz**, you can determine the contents of the symbolic link by using either the `lstat` function or the `readlink` function with a 0 value for **bufsiz**.

### Error Return

If the `readlink` function is not successful, it returns a value of `-1` and sets `errno` to one of the following:

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

#### **ENAMETOOLONG**

**path** is longer than `PATH_MAX` characters or some component of **pathname** is longer than `NAME_MAX` characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds `PATH_MAX`.

**ENOENT** The named file does not exist.

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

**ELOOP** A loop exists in symbolic links. This error is issued if the number of symbolic links found while resolving the **path** argument. is greater than `POSIX_SYMLINK_MAX`.

**EINVAL** The named file is not a symbolic link.

## readlink

# Programming Considerations

None.

## Examples

The following example provides readlink information for a file.

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

main() {
    char fn[]="readlink.file";
    char sl[]="readlink.symlink";
    char buf[30];
    int fd;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        if (symlink(fn, sl) != 0)
            perror("symlink() error");
        else {
            if (readlink(sl, buf, sizeof(buf)) < 0)
                perror("readlink() error");
            else printf("readlink() returned '%s' for '%s'\n", buf, sl);

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

### Output

readlink() returned 'readlink.file' for 'readlink.symlink'

## Related Information

- “lstat–Get Status of a File or Symbolic Link” on page 317
- “symlink–Create a Symbolic Link to a Path Name” on page 521
- “unlink–Remove a Directory Entry” on page 668.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



## rehka–Rehook Core Block

The function transfers the information from an 8-byte field to a specified core block reference word (CBRW) in the ECB, and zeros the 8-byte field. This allows access to the specified block by successive ECBs. If the field specified is in global storage, the `glob` function is used to get the address of the specified global, and the `global` function is used to update the field. The core block referred to by the 8-byte field must have been *unhooked* by this function.

## Format

```
#include <tpfapi.h>
void rehka(enum t_lvl level, enum t_hook_type glob_indicator, ...);
```

### level

One of 16 possible values representing a valid data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This argument represents the CBRW from where information is to be transferred from.

### glob\_indicator

This argument must belong to enumeration type `t_global` specifying 1 of 2 possible values:

#### REHKA\_GLOBAL

Specifies that the save area is in protected storage.

#### REHKA\_UNPROTECTED

Specifies that the save area is *not* in protected storage.

... This argument is either an address if the 8-byte save area is not in protected storage or a global tag if the field is in protected storage.

## Normal Return

Void.

## Error Return

Not applicable.

## Programming Considerations

- The location of the last parameter will contain the contents of the CBRW specified by `CE1CR(literal)`.
- The use of this function must be limited to those functions having unique buffering requirements that demand a specific block be capable of access by successive ECBs. It is *not* to be used to obtain use of a *free* data level.
- The data level specified must not have an attached block.
- The `rehka` function may use the `global` function.
- If the save area supplied is a synchronizable global field that needs to be locked before modified, it then becomes the user's responsibility to call the `global` function with the `GLOBAL_LOCK`, `GLOBAL_UNLOCK` or `GLOBAL_SYNC` option, appropriately.
- If an incorrect parameter is given, a system error with `exit` occurs.

## Examples

The following example rehooks a core block to level 0 from a location pointed to by `save_ptr`.

## rehka

```
#include <tpfapi.h>
:
char    save_area[8];
:
rehka(D0, REHKA_UNPROTECTED, save_area);
:
```

The following example rehooks a core block to level 0 from a location specified by the global tag `_s2hok`.

```
#include <tpfapi.h>
:
rehka(D0, REHKA_PROTECTED, _s2hok);
:
```

## Related Information

“unhka–Unhook Core Block” on page 666.

## relcc—Release Working Storage Block

This function returns a working storage block to the system.

### Format

```
#include <tpfapi.h>
void relcc(enum t_lvl level);
```

or

```
#include <tpfapi.h>
void relcc(TPF_DECB *decb);
```

#### level

One of 16 possible values representing a valid entry control block (ECB) data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter identifies the core block reference word (CBRW) containing the address of the working storage block to be returned to the system.

#### decb

A pointer to a data event control block (DECB). This parameter identifies the CBRW containing the address of the working storage block to be returned to the TPF system.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The specified ECB data level or DECB must be occupied with a valid working storage block address and other control information when the function is called; otherwise, a system error with exit results.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example tests for, then releases, a working storage block on level DB.

```
#include <tpfapi.h>
:
:
if (levtest(DB))
    relcc(DB);
```

The following example tests for, and then releases, a working storage block held in a DECB.

```
#include <tpfapi.h>
:
:
TPF_DECB *decb;
:
:
if (levtest(decb))
    relcc(decb);
```

**relcc**

## **Related Information**

- “crusa—Free Core Storage Block If Held” on page 73
- “detac\_id—Detach a Working Storage Block from the ECB” on page 88
- “getcc—Obtain Working Storage Block” on page 248
- “tpf\_rcrhc—Release a Core Block and File Address” on page 624.

See *TPF Application Programming* for more information about DECBs.

## relfc—Release File Pool Storage

This function returns a pool file address to the system.

### Format

```
#include <tpfio.h>
void      relfc(enum t_lvl level);
```

or

```
#include <tpfio.h>
void      relfc(TPF_DECB *decb);
```

#### level

One of 16 possible values representing a valid entry control block (ECB) data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter identifies the file address reference word (FARW) containing the pool file address to be returned to the system.

#### decb

A pointer to a data event control block (DECB). This parameter identifies the FARW containing the pool file address to be returned to the TPF system.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- There must be a valid pool file address in the FARW at the specified ECB data level or DECB.
- No record address may be released more than once.
- TPF transaction services processing affects `relfc` processing in the following ways:
  - File addresses are released only at commit time.
  - When rollback occurs, file address release requests are discarded and the file addresses are not released.
  - If a system error occurs, processing ends as if a rollback was issued.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example releases all forward chain pool records from a message block and zeros the forward chain field in the prime message block (prime message block already on level 1).

```
#include <tpfio.h>
#include <c$am0sg.h>

struct am0sg *prime,*chain;          /* Pointers to message blocks */
```

## relfc

```
unsigned int adrs[100];
int j,i = 0;

prime = ecbptr()->celcr1;          /* Base prime message block */
chain = prime;

while(chain->am0fch != 0)          /* Obtain all chain addresses */
{
    adrs[i] = chain->am0fch;
    crusa(D2);                    /* Release last core block */
    chain = find_record(D2,&adrs[i++], "OM", '\0', NOHOLD);
}
for(j = 0; j < i; j++)            /* Release all chain addresses */
{
    ecbptr()->celfm2 = adrs[j];    /* Set up FARW */
    relfc(D2);                    /* and release the address */
}
```

The following example releases a pool record that is held in the DECB named POOLREC.

```
#include <tpfio.h>
TPF_DECB *decb;
DECBC_RC rc;
char decb_name[16] = "POOLREC";
:
:
if ((decb = tpf_decb_locate(decb_name, &rc)) != NULL)
{
    relfc(decb);
}
else
{
    /* error */
}
```

## Related Information

- “getfc—Obtain File Pool Address” on page 258
- “tpf\_rcrfc—Release a Core Block and File Address” on page 624.

See *TPF Application Programming* for more information about DECBs.

## relpc—Release Program from Core Lock

This function is used to unlock a file resident program from core.

### Format

```
#include <tpfapi.h>
int      relpc(const char *name, int unlock, int error,
               const char *loadset);
```

#### name

A pointer to the character representation of the program name. The name is 4 alphanumeric characters that should have been allocated at system generation time. The program must be nonprivate.

#### unlock

This argument is an integer describing whether the program was locked with a SPECIAL request. Use the defined terms **RELPC\_UNLOCK** to unlock a program that was not locked with a SPECIAL request or **RELPC\_SPECIAL** to unlock a program that was locked with a SPECIAL request.

#### error

This argument is an integer describing whether control is returned to the caller when an error occurs or if an OPR-066 should be issued. Use the defined terms **RELPC\_NODUMP** to denote that control should be returned to the caller, or **RELPC\_DUMP** to denote that the service routine should issue an OPR-066. The possible errors are: invalid program name, valid program name but invalid loadset name, the program is allocated as private, or **RELPC\_SPECIAL** was requested and the special indicator is not set.

#### loadset

A pointer to a valid loadset name, or BASE if referring to the base allocated program. If this parameter is used, the active program that is associated with the specified loadset is affected. If this parameter is specified as NULL, the version of the program associated with the requesting ECB is affected.

### Normal Return

0

### Error Return

A nonzero error return indicates that the function did not complete successfully.

Table 11. relpc Error Return

Numbers	Return Code	Description
#define RELPC_SNAPC_ERR	4	
#define RELPC_LS_NF	8	LOADSET VERSION NOT FOUND
#define RELPC_SP_NOTSET	16	SPECIAL LOCK NOT/ALREADY SET
#define RELPC_PRIVATE	64	PROGRAM IS A PRIVATE PROGRAM
#define RELPC_PGM_NF	128	PROGRAM NOT FOUND

### Programming Considerations

- relpc should only be issued for programs that have been previously locked in core via the getpc function.
- If the getpc function was issued multiple times against a specific program, it requires an equal number of relpc calls to unlock the program from core.

## relpc

- **RELPC\_SPECIAL** must be specified on the **unlock** parameter of the relpc function if **GETPC\_SPECIAL** was specified on the **lock** parameter of the getpc function.

## Examples

The following example unlocks program QZZ5 in loadset OCTOBER from core. When QZZ5 was locked in core via getpc, a SPECIAL request was indicated. If an error occurs, the service routine issues an OPR-066.

```
#include <tpfapi.h>
:
:
relpc("QZZ5",RELPC_SPECIAL,RELPC_DUMP,"OCTOBER");
:
```

## Related Information

“getpc—Get Program and Lock in Core” on page 265.



---

## remove—Delete a File

This function deletes a link to a file.

### Format

```
#include <stdio.h>
int remove(const char *filename);
```

**filename**

The path name of the link to be deleted.

### Normal Return

If successful, the remove function returns a 0 value.

### Error Return

If not successful, the remove function returns a nonzero value.

### Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

### Examples

In the following example, the remove function is called with a file name. The program issues a message if an error occurs.

```
#include <stdio.h>

int main(int argc, char ** argv)
{
    if ( argc != 2 )
        printf( "Usage: %s fn\n", argv[0] );
    else
        if ( remove( argv[1] ) != 0 )
            printf( "Could not remove file\n" );
}
```

### Related Information

- “fopen—Open a File” on page 199
- “rename—Rename a File” on page 428.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## rename—Rename a File

This function renames a file.

### TPF deviation from POSIX

The TPF system does not support the `ctime` (change time) time stamp.

## Format

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

### **oldname**

The name by which the file is currently known.

### **newname**

The name by which the file will now be known.

This function changes the name of the file from the name pointed to by **oldname** to the name pointed to by **newname**.

The **oldname** pointer must point to the name of an existing file.

Both **oldname** and **newname** must be of the same type; that is, both directories or both files.

If **newname** already exists, it is removed before **oldname** is renamed to **newname**. Therefore, if **newname** specifies the name of an existing directory, it must be an empty directory.

If the **oldname** argument points to a symbolic link, the symbolic link is renamed. If the **newname** argument points to a symbolic link, the link is removed and **oldname** is renamed to **newname**. The rename function does not affect any file or directory named by the contents of the symbolic link.

For the rename function to be successful, the process needs write permission on the directory containing **oldname** and the directory containing **newname**. If **oldname** and **newname** are directories, the rename function also needs write permission on the directories themselves.

If **oldname** and **newname** both refer to the same file, rename returns successfully and performs no other action.

When the rename function is successful, it updates the modification times for the parent directories of **oldname** and **newname**.

## Normal Return

The rename function returns a value of 0 if it is successful.

## Error Return

If not successful, the rename function returns a nonzero value and sets `errno` to one of the following:

**EACCES**      An error occurred for one of the following reasons:

- The process did not have search permission on some component of the old or new path name.
- The process did not have write permission on the parent directory of the file or directory to be renamed.
- The process did not have write permission on **oldname** or **newname**.

**EBUSY** **oldname** and **newname** specify directories, but one of them is the root directory (/).

**EINVAL** This error occurs for one of the following reasons:

- **oldname** is part of the path name prefix of **newname**.
- **oldname** or **newname** refer to either dot (.) or dot-dot (..).

**EISDIR** **newname** is a directory but **oldname** is not a directory.

**ELOOP** A loop exists in symbolic links. This error is issued if the number of symbolic links found while resolving the **oldname** or **newname** argument is greater than POSIX\_SYMLINK\_MAX.

**ENAMETOOLONG** **pathname** is longer than PATH\_MAX characters or some component of **pathname** is longer than NAME\_MAX characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH\_MAX.

**ENOENT** No file or directory named **oldname** was found, or either **oldname** or **newname** was not specified.

**ENOSPC** The directory intended to contain **newname** cannot be extended.

**ENOTDIR** A component of the pathname prefix for **oldname** or **newname** is not a directory, or **oldname** is a directory and **newname** is a file that is not a directory.

**ENOTEMPTY** **newname** specifies a directory, but the directory is not empty.

**ETPFNPIPWSYS** In a loosely coupled environment, there was an attempt to access a FIFO special file (or named pipe) from a processor other than the processor on which the file was created.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

Because the rename function requires an update to at least one directory file, files cannot be renamed in 1052 or UTIL state.

## Examples

The following example takes two file names as input and uses rename to change the file name from the first name to the second name.

```
#include <stdio.h>

int main(int argc, char ** argv )
{
    if ( argc != 3 )
```

## rename

```
        printf( "Usage: %s old_fn new_fn\n", argv[0] );
    else if ( rename( argv[1], argv[2] ) != 0 )
        printf( "Could not rename file\n" );
}
```

## Related Information

- “mkfifo—Make a FIFO Special File” on page 328
- “remove—Delete a File” on page 427.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## rewind—Set File Position to Beginning of File

This function sets file position to the beginning of the file.

### Format

```
#include <stdio.h>
void rewind(FILE *stream);
```

#### **stream**

The stream to be rewound.

This function repositions the file position indicator of the stream pointed to by **stream**. A call to the rewind function is the same as the statement that follows except that rewind also clears the error indicator for the **stream**.

```
(void) fseek(stream, 0L, SEEK_SET);
```

### Normal Return

There is no returned value.

### Error Return

If an error occurs, `errno` is set. After the error, the file position does not change. The next operation can be either a read or a write operation.

## Programming Considerations

None.

## Examples

The following example first opens a `myfile.dat` file for input and output. It writes integers to the file, uses the `rewind` function to reposition the file pointer to the beginning of the file, and then reads back the data.

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    int data1, data2, data3, data4;
    data1 = 1; data2 = -37;

    /* Place data in the file */
    stream = fopen("myfile.dat", "w+");
    fprintf(stream, "%d %d\n", data1, data2);

    /* Now read the data file */
    rewind(stream);
    fscanf(stream, "%d", &data3);
    fscanf(stream, "%d", &data4);
    printf("The values read back in are: %d and %d\n",
        data3, data4);
}
```

#### **Output**

The values read back in are: 1 and -37

## Related Information

- “`fgetpos`—Get File Position” on page 148
- “`fseek`—Change File Position” on page 226

## rewind

- “fsetpos–Set File Position” on page 228
- “ftell–Get Current File Position” on page 234.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## rewinddir—Reposition a Directory Stream to the Beginning

This function repositions an open directory stream to the beginning of the stream.

### Format

```
#include <dirent.h>
void rewinddir(DIR *dir);
```

#### **dir**

A pointer to the DIR object of the open directory to be rewound.

The next call to the `readdir` function reads the first entry in the directory. If the contents of the directory have changed since the directory was opened, a call to the `rewinddir` function updates the directory stream so that a subsequent `readdir` function can read the new contents.

### Normal Return

There is no returned value.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example produces the contents of a directory by opening it, rewinding it, and closing it.

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

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

    if ((dir = opendir("/")) == NULL)
        perror("opendir() error");
    else {
        puts("contents of root:");
        while ((entry = readdir(dir)) != NULL)
            printf("%s ", entry->d_name);
        rewinddir(dir);
        puts("");
        while ((entry = readdir(dir)) != NULL)
            printf("%s ", entry->d_name);
        closedir(dir);
        puts("");
    }
}
```

#### **Output**

```
contents of root:
. .. dev usr
. .. dev usr
```

**rewinddir**

## **Related Information**

- “closedir—Close a Directory” on page 46
- “opendir—Open a Directory” on page 384
- “readdir—Read an Entry from a Directory” on page 415.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



## ridcc–SNA RID Conversions

This function returns the following configuration information about an SNA resource:

- RID or SCBID
- RVT1 or SCB1 address
- RVT2 or SCB1 address
- Subarea address table (SAT) address
- Network address
- Pseudo LNIATA.

## Format

```
#include <sysapi.h>
struct ridcc_return *ridcc(enum ridcc_func func, int id);
```

### func

One of the following values that describes the type of input argument:

#### RIDCC\_RID

Resource identifier (RID) or session control block identifier (SCBID)

#### RIDCC\_NA

Network address.

#### RIDCC\_RVT1

Resource vector table part 1 (RVT1) address or session control block part 1 (SCB1) address

#### RIDCC\_RVT2

Resource vector table part 2 (RVT2) address or session control block part 2 (SCB2) address

#### RIDCC\_PLIT

Pseudo line number, interchange address, and terminal address (LNIATA).

**id** Value of the input argument specified in the **func** parameter, passed as an integer type.

## Normal Return

Pointer to a structure of type `ridcc_return`, which is defined in `sysapi.h`.

## Error Return

NULL pointer.

## Programming Considerations

- One return area for each I-stream is allocated in the CP. The returned data should be saved by the calling segment before issuing any macro that could cause it to give up control; once control has passed to another ECB, the contents of the return area cannot be predicted.
- The programmer must be aware of the size of the input arguments:

Argument	Size
RID or SCBID	3 bytes
RVT1 or SCB1	4 bytes
RVT2 or SCB2	4 bytes

## ridcc

<b>Network address</b>	3 bytes
<b>Pseudo LNIATA</b>	3 bytes.

## Examples

The following example converts an RID to the RVT1 address.

```
#include <sysapi.h>
:
:
int rid;
void *rvt1_address;
struct ridcc_return *ridcc_parm;
:
rid = ecbptr()->ebrout
if (ridcc_parm = ridcc(RIDCC_RID,rid))
    rvt1_address = ridcc_parm->ridrvt1;
```

## Related Information

None.

## rlcha—Release Chained File Records

This function releases chained pool record addresses from the record first specified through the end of chain using standard TPF record headers. The input argument **hdr** may point at any standard header. The record ID and the record code check for all records in the chain must match the header pointed to by **hdr**.

### Format

```
#include <tpfapi.h>
void      rlcha(struct stdhdr *hdr);
```

or

```
#include <tpfapi.h>
#include <c$std8.h>
void      rlcha(struct istd8 *hdr);
```

**hdr**

This argument is a pointer to struct stdhdr (see tpfapi.h) or struct istd8 (see c\$std8.h), which describe the TPF standard record header.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- To be released, all succeeding records must have the same record ID and code check as the first record in the chain. If the succeeding records do not have the same ID and code check, the RLCH or CLC8 subroutine exits.
- TPF transaction services processing affects rlcha processing in the following ways:
  - Delay issuing the rlcha function until commit processing has completed successfully.
  - When rollback occurs, rlcha function requests are discarded and the file addresses are not released.
- Applications that call this function using the struct istd8 interface must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example starts the release of the chain of pool record addresses beginning with the address found in the standard header on level D5.

```
#include <tpfapi.h>
struct stdhdr *cp0hdr;
:
cp0hdr = ecbptr()->ce1cr5;
rlcha(cp0hdr);
```

## rlcha

The following example starts the release of the chain of pool record addresses beginning with the address found in the 8-byte standard header on the DECB pointed to by **decb\_ptr**.

```
#include <tpfapi.h>
#include <istd8.h>
#include <i$decb.h>

TPF_DECB *decb_ptr;
DECBC_RC rc;
struct istd8* block_header

decb_ptr = tpf_decb_create( NULL, &rc );
:
:
block_header = (istd8 *)decb_ptr->idecdad;
rlcha( block_header );
```

## Related Information

- “getfc—Obtain File Pool Address” on page 258
- “relfc—Release File Pool Storage” on page 423.

## rmdir—Remove a Directory

This function removes a directory.

### TPF deviation from POSIX

The TPF system does not support the `ctime` (change time) time stamp.

## Format

```
#include <unistd.h>
int rmdir(const char *pathname);
```

### pathname

A pointer to the path name of the directory to be deleted.

This function removes a directory, **pathname**, provided that the directory is empty. **pathname** must not end in dot (.) or dot-dot (..).

The `rmdir` function does not remove a directory that still contains files or subdirectories.

If **pathname** refers to a symbolic link, the `rmdir` function fails and sets `errno` to `ENOTDIR`.

If no process currently has the directory open, the `rmdir` function deletes the directory itself. The space occupied by the directory is freed for new use. If one or more processes have the directory open when it is removed, the directory remains accessible to those processes. New files cannot be created under a directory after the last link is removed even if the directory is still open.

The `rmdir` function removes the directory even if it is the working directory of a process.

If a `rmdir` function is successful, the modification time for the parent directory is updated.

## Normal Return

If successful, the `rmdir` function returns a value of 0.

## Error Return

If unsuccessful, the `rmdir` function returns a value of `-1` and sets `errno` to one of the following:

<b>EACCES</b>	The process did not have search permission for some component of <b>pathname</b> or it did not have write permission for the directory containing the directory to be removed.
<b>EBUSY</b>	<b>pathname</b> cannot be removed because it is currently being used by the system or a process.
<b>EINVAL</b>	The last component of <b>pathname</b> contains a dot (.) or a dot-dot (..).
<b>ELOOP</b>	A loop exists in symbolic links. This error is issued if the number of symbolic links found while resolving the <b>pathname</b> argument is greater than <code>POSIX_SYMLLOOP</code> .

## rmkdir

### ENAMETOOLONG

**pathname** is longer than PATH\_MAX characters or some component of **pathname** is longer than NAME\_MAX characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH\_MAX.

### ENOENT

**pathname** does not exist or it is an empty string.

### ENOTDIR

Some component of the **pathname** prefix is not a directory.

### ENOTEMPTY

The directory still contains files or subdirectories.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

Because the `rmkdir` function requires an update to a directory file and the deletion of a second directory file, directories cannot be removed in 1052 or UTIL state.

## Examples

The following example removes a directory.

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

main() {
    char new_dir[]="new.dir";
    char new_file[]="new.dir/new.file";
    int fd;

    if (mkdir(new_dir, S_IRWXU|S_IRGRP|S_IXGRP) != 0)
        perror("mkdir() error");
    else if ((fd = creat(new_file, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        unlink(new_file);
    }

    if (rmdir(new_dir) != 0)
        perror("rmdir() error");
    else
        puts("removed!");
}
```

## Related Information

- “mkdir—Make a Directory” on page 326
- “unlink—Remove a Directory Entry” on page 668.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## routc—Route a Message

This function routes a message to a terminal or to another application program.

### Format

```
#include <tpfio.h>
#include <c$rc0pl.h>
void      routc(struct rc0pl *rcpl, enum t_lvl level);
```

#### rcpl

This argument is a pointer to type `struct rc0pl`, included in `tpfapi.h`, that defines a routing control parameter list (RCPL).

#### level

One of 16 possible values representing a valid data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This argument identifies the CBRW level where the working storage block containing the message to be transmitted is.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The first or only segment of the message to be routed must be contained in a working storage block on the referenced level. If the message is not completely contained in the storage block, it should be continued in 1 or more long-term pool records of the same size. For messages sent to terminals, only 381-byte storage blocks are allowed.
- Messages sent to application programs must be in application message format.
- Messages sent to terminals may be either in application message format or in output message format. The contents of the RCPL indicate which format is in use.

**Note:** The structure `am0sg`, which defines an application message block, is defined in `c$am0sg.h`.

- On return, the working storage block on the specified level is released to the system and made unavailable to the operational program.
- If the ROUTC service routine detects the following error conditions, a system error with exit results.
  - No main storage block on the specified CBRW level
  - Invalid message length
  - Invalid contents of the message block and the RCPL or both.

## Examples

The following example calls `routc` to send a message to the originating user terminal, because the message block has already been built on level `D4` and the RCPL remains at `EBW000`.

## putc

```
#include <tpfio.h>
#include <c$rc0pl.h>
...
putc((struct rc0pl *)&(ecbptr()->ebw000),D4);
```

## Related Information

“puts—Put String to Standard Output Stream” on page 402.



## rsysc—Release Storage from the System Heap

This function releases storage from the system heap, and returns the 4-KB frames to the pool of available frames.

### Format

```
#include <sysapi.h>
int rsysc(void *address, int frames, char *token);
```

#### address

This argument specifies the address of the storage to be released from the system heap.

#### frames

This argument specifies the number of 4-KB frames to be released.

#### token

This argument specifies a pointer to the string that was specified when the storage was acquired.

### Normal Return

RSYSC\_OK indicates that the storage is released successfully.

### Error Return

If the function was passed an incorrect address, number of frames, or token, error code RSYSC\_ERROR is returned and no storage is released.

## Programming Considerations

- Any storage that is acquired by `gsysc()` must be returned by the application using `rsysc()` because the storage is not attached to the ECB and will not be returned when the ECB exits.
- Storage acquired by `gsysc()` is tagged with the string specified by the `token` parameter. You must specify the same string when you release the storage using `rsysc()`.
- The application that acquires system heap storage needs to save the starting address of the storage, the number of frames allocated, and the token specified.

## Examples

The following example allocates and releases 12 KB of system heap storage.

```
#include <sysapi.h>
#include <stdio.h>
...
{
    /******
    /* Allocate 12 KB of storage for a table used by many different    */
    /* ECBs.                                                         */
    /******
    int frames, rc;
    char * token = "TABLE40 ";
    struct table {
        char *name;
        int code;
    } *tbl_ptr;

    frames = 3;
    tbl_ptr = gsysc(frames, token);
    if (tbl_ptr == 0) {
```

## rsysc

```
        serrc_op(SERRC_EXIT, 0x1111, "Error allocating table.", NULL);
    }
    :
    rc = rsysc((void *)tbl_ptr, frames, token);
    if (rc == RSYSC_ERROR) {
        serrc_op(SERRC_EXIT, 0x2222, "Error releasing storage.", NULL);
    }
}
```

## Related Information

“gsysc—Get Storage from the System Heap” on page 293.

## rvtcc–Search Resource Vector Table (RVT) Entries

This function searches the RVT for the next entry that matches the search criteria specified in the input arguments.

### Format

```
#include <sysapi.h>
struct rvtcc_return *rvtcc(enum rvtcc_step step, enum rvtcc_arg arg,
                           const void *rvt1, const void *rvt2,
                           struct rvtcc_return *rvt_s);
```

#### step

One of the following step types:

##### **RVTCC\_NEXT**

Returns the next resource in the resource vector table (RVT)

##### **RVTCC\_PREV**

Returns the previous resource in the RVT.

#### arg

One of the following search types:

##### **RVTCC\_CCW**

When step=RVTCC\_NEXT, the RVT entry for the next resource with a session using the ALS/NCP/CTC is returned. When step=RVTCC\_PREV, the RVT entry for the ALS/NCP/CTC is returned.

##### **RVTCC\_CDRM**

When step=RVTCC\_NEXT, the RVT entry for the next resource owned by the CDRM is returned. When step=RVTCC\_PREV, the RVT entry for the CDRM is returned.

##### **RVTCC\_LOCAL**

When step=RVTCC\_NEXT, the RVT entry for the next local resource is returned. When step=RVTCC\_PREV, the entry for the local system services control point (SSCP) is returned if the input entry is not a secondary logical unit (SLU) thread. If the input entry is an SLU thread, the entry for the application primary logical unit (PLU) is returned.

##### **RVTCC\_NONE**

When step=RVTCC\_NEXT, the RVT entry for the next resource in collating sequence is returned. When step=RVTCC\_PREV, the entry for the previous resource in collating sequence is returned.

#### rvt1

A pointer to a resource vector table part 1 (RVT1) or session control block part 1 (SCB1) entry that uniquely identifies an SNA resource or a NULL pointer.

#### rvt2

A pointer to a resource vector table part 2 (RVT2) or session control block part 2 (SCB2) entry that uniquely identifies an SNA resource or a NULL pointer.

#### rvt\_s

A pointer to storage allocated for the structure rvtcc\_return.

### Normal Return

Pointer to a structure of type rvtcc\_return, which is defined in sysapi.h.

### Error Return

NULL pointer.

**rvtcc**

## Programming Considerations

Either the rvt1 or rvt2 input argument must contain a valid entry address. The other argument must be NULL. If both arguments are not NULL, the rvt1 argument is used. No error checking is performed to verify that both pointers are consistent.

## Examples

The following example locates the next resource owned by a specific CDRM.

```
#include <sysapi.h>
:
void *rvt1_address;
struct rvtcc_return &rvtcc_list,
:               *rvtcc_parm=&rvtcc_list;
:
rvtcc_parm = rvtcc_return (RVTCC_NEXT, RVTCC_CDRM, rvt1_address, NULL,
:               rvtcc_list);
```

## Related Information

None.

---

## scanf—Scan Input for Variables

The information for this function is included in “fscanf, scanf, sscanf—Read and Format Data” on page 217.

## selec–Select a Thread Application Interface

This function is used to obtain an RCPL or session partner name.

### Format

```
#include <tpfapi.h>
union selec_return *selec(int req_type, union selec_arg *s_req);
```

#### req\_type

Code the defined terms **SELEC\_NAME** if a network qualified name (NQN) is to be used as a search argument, or **SELEC\_RCPL** if an RCPL is to be used as a search argument.

#### s\_req

Pointer to union of type `selec_arg`, which may contain either a structure of type `name_type` or `rcpl_type` containing the NQN or RCPL. This union is defined in `tpfapi.h`.

### Normal Return

Pointer to union of type `selec_return`, which may contain either a structure of type `selec_ret_name` or `selec_ret_RCPL` containing the requested information. This union is defined in `tpfapi.h`.

### Error Return

The `selrtncd` field in `selec_return` is used to return errors discovered in the `selec_arg` parameters. The following codes are defined:

- 4** Invalid name; bad return from the INQRC macro
- 8** Invalid request
  - SLU not in session with PLU
  - Not a PLU/SLU combination.
- 12** Database error; bad return from RIDCC macro
- 16** SLU not bound, or not in session
- 20** Invalid request
  - Origin/destination. Not cross-domain environment.
  - Origin/destination. Not an LU.
- 32** SLU not active
- 36** Discrepancy between input qualifier and input SLU
- 40** Qualifier invalid; does not point to SLU.

## Programming Considerations

If **req\_type** is not **SELEC\_NAME** or **SELEC\_RCPL**, a system error with exit results.

### Examples

The following example calls `selec` to obtain the RCPL for the network qualified name at EBW032. A check is performed for certain errors.

```
#include <tpfapi.h>
union selec_arg *reqptr;
union selec_return *retptr;
:
reqptr = (union selec_arg *) &(ecbptr()->ebw032); /* storage for selec_arg */
```

```
strncpy(&(reqptr->name_type.selorg), "TPFDATA ", 8); /* fill in blanks for */
strncpy(&(reqptr->name_type.selnetid), "MVS1 ", 8); /* length of 8 */
strncpy(&(reqptr->name_type.selnode), "DB2 ", 8);
retptr = selec(SELEC_NAME, reqptr);

switch (retptr->selec_ret_name.selrtncd)
{
    case 04: serrc_op(SERRC_EXIT, 0x999004, "Invalid Name", NULL);
    case 32: serrc_op(SERRC_EXIT, 0x999020, "SLU not active", NULL);
}
```

## Related Information

"inqrc—Convert Resource Application Interface" on page 295.

## select—Monitor Read, Write, and Exception Status

The `select` function monitors a list of file descriptors for readability, readiness for writing, and exception pending conditions. The list can contain nonsocket file descriptors, socket descriptors, or a combination of both.

### Format

```
#include <socket.h>
int      select(int *s,
                short int noreads,
                short int nowrites,
                short int noexcepts,
                long int timeout);
```

- s** Pointer to an array containing a list of file descriptors to check for readability, followed by a list of file descriptors to check for readiness for writing, followed by a list of file descriptors to check for exception pending conditions. The array can contain nonsocket file descriptors, socket descriptors, or a combination of both.

**noreads**

Number of file descriptors to be checked for readability.

**nowrites**

Number of file descriptors to be checked for readiness for writing.

**noexcepts**

Number of file descriptors to be checked for exception pending conditions.

**timeout**

Maximum interval, in milliseconds, to wait for the selection to be completed.

If the timeout value is 0, `select` does not wait before returning to the caller. If the timeout value is `-1`, `select` does not time out, but it returns when a file descriptor becomes ready. If the timeout value is a number of milliseconds, `select` waits for the specified interval before returning to the caller.

### Normal Return

The total number of ready file descriptors. A value of 0 indicates an expired time limit. If the return value is greater than 0, the file descriptors in the `s` argument that were not ready are set to `-1`.

### Error Return

A value of `-1`.

If an error occurs while monitoring a socket descriptor, `errno` and `sock_errno` are set to one of the following error codes. Unless otherwise stated in the description, the following error codes can be returned for either TCP/IP offload support or TCP/IP native stack support.

**SOCFAULT**

One of the following occurred:

- The address is not valid. This only applies to TCP/IP offload support.
- A timeout value was specified that is not valid.

**SOCNOTSOCK**

One of the sockets descriptors specified in the file descriptor array is not valid.



<b>E1052STATE</b>	The socket was closed because the system was in or cycling down to 1052 state.
<b>EINACT</b>	All offload devices associated with the socket descriptor have been disconnected. The socket is closed. This error code is returned only for TCP/IP offload support.
<b>EINACTWS</b>	An offload device associated with the socket descriptor has been disconnected. The socket is still available. This error code is returned only for TCP/IP offload support.
<b>ESYSTEMERROR</b>	A system error has occurred and closed the socket. This error code is returned only for TCP/IP offload support.
<b>EIBMIUCVERR</b>	An error occurred while the function call was sent to the offload device. This error code is returned only for TCP/IP offload support.

If an error occurs while monitoring a nonsocket file descriptor, `errno` is set to the following:

<b>EBADF</b>	One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor or a file descriptor that is not supported through the <code>select</code> function.
--------------	---

## Programming Considerations

- The `select` function monitors activity on a set of various file descriptors to verify if any of the file descriptors are ready for reading or writing, or if any exceptional conditions are pending. If the time limit expires, `select` returns to the caller.
- If any file descriptors specified in the **s** argument represent sockets, you *must* issue a `bind` function on all the file descriptors you are monitoring that are sockets before issuing the `select` function.
- If any file descriptors specified in the **s** argument represent sockets, all the socket descriptors passed to the `select` function must be using TCP/IP offload support or all the socket descriptors must be using TCP/IP native stack support. You cannot monitor a mixture of sockets using TCP/IP offload support and TCP/IP native stack support on the same `select` function call.
- For file descriptors specified in the **s** argument that represent sockets using TCP/IP native stack support, the send low-water mark determines the minimum amount of data that must be available in the send buffer to allow the `select` for a write function to be processed. The send low-water mark is set by the `SO_SNDLOWAT` socket option of the `setsockopt` function.
- You cannot specify nonsocket file descriptors and socket descriptors using TCP/IP offload support on the same `select` function call.
- The sum of the numbers specified for the **noreads**, **nowrites**, and **noexcepts** arguments cannot exceed the number of file descriptors specified in the **s** argument.
- The `select` function takes a single array of file descriptors as input. If your application uses the BSD `select` model instead, use the `tpf_select_bsd` function rather than the `select` function.

## Examples

The following example monitors a FIFO special file (or named pipe) and a socket for incoming messages.

## select

```
#define MSG_BUFF 4096
#define TIMERINT 30
#define MAX_SELECT_FD 20
#define SELECT_TIMEOUT 2*1000

#include <sys/socket.h>
#include <sys/signal.h>
#include <stdio.h>
#include <fcntl.h>

void SigAlrmH(int SIG_TYPE)    /* Handle alarm */

{
    char incoming_data[MSG_BUFF + 1];
    int bytes_in, i;

    int fd_fifo;                /* mkfifo file descriptor */
    char fifopath[] = "/tmp/my_fifo";

    int sd_inet;                /* socket descriptor */
    int slen;
    struct sockaddr_in sin, sockinet;

    int select_array[MAX_SELECT_FD];
    int nfds, Rfd, Wfd, Xfd;

    signal(SIGALRM, SigAlrmH);
    alarm(TIMERINT);

    /* Open the named pipe. */
    if ((mkfifo(fifopath, 777)) < 0)
        /* Handle the mkfifo error. */

    if ((fd_fifo = open(fifopath, O_RDONLY, 0)) < 0)
        /* Handle the open error. */

    /* Open the socket. */
    if ((sd_inet = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        /* Handle socket error. */
    /* Set sockaddr parameters. */
    if ((bind(sd_inet, (struct sockaddr *)&sin, sizeof(sin))) < 0)
        /* Handle bind error. */

    for (;;) {
        /* Initialize select Parameters */
        select_array[0] = fd_fifo;
        select_array[1] = sd_inet;
        Rfd = 2;
        Wfd = 0;
        Xfd = 0;

        /* Enable signal interrupts to accept any timeout alarm. */
        tpf_process_signals();

        /* Wait for incoming data, select timeout, or external alarm. */
        nfds = select(select_array, Rfd, Wfd, Xfd, SELECT_TIMEOUT);
        if (nfds == 0)    /* Process the select timeout condition. */
        if (nfds < 0)    /* Process the error condition. */

        /* Process the readable file or socket descriptors */
        for (i=0; i<nfds; i++) {
            if (select_array[i] == fd_fifo ) {
                bytes_in = read(fd_fifo, incoming_data, MSG_BUFF);
                /* Process the data from the named pipe. */
            }
            if (select_array[i] == sd_inet) {
                slen = sizeof sockinet;
            }
        }
    }
}
```

```

        bytes_in = recvfrom(sd_inet, incoming_data, MSG_BUFF, 0,
                           (struct sockaddr *) &sockinet, &slen);
        /* Process the socket data. */
    }
} /* end for loop */
} /* end for loop */
}

```

## Related Information

- “mkfifo—Make a FIFO Special File” on page 328
- See the following related functions in *TPF Transmission Control Protocol/Internet Protocol*:
  - accept
  - connect
  - read
  - send
  - setsockopt
  - sock\_errno
  - “tpf\_select\_bsd—Indicates Read, Write, and Exception Status” on page 629.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about nonsocket file descriptors. See *TPF Transmission Control Protocol/Internet Protocol* for more information about socket file descriptors.

## serrc\_op—Issue System Error: Operational

This function causes a storage dump to be issued by the system error routine, and may optionally display a message at prime CRAS. The entry control block (ECB) may be exited, if desired.

### Format

```
#include <tpfapi.h>
void    serrc_op(enum t_serrc status, int number, const char *msg,
               void *slist[]);
```

#### status

The status of the ECB following the dump. This argument must belong to enumeration type `t_serrc`, which is defined in `tpfapi.h`. Code the defined term **SERRC\_EXIT** to force the ECB to exit, **SERRC\_RETURN** to cause a return to the calling program, or **SERRC\_CATA** to cause a catastrophic error.

#### number

The identification number for the dump. This argument is an integer and should be a unique number ranging from 1 to `X'FFFFFF'`. This number is prefixed with a U. If you want to control the character that is used as a prefix, use `serrc_op_ext`.

#### msg

This argument is a pointer to type `char`, which is a message text string to be displayed at the CRAS console and appended to the dump. This string must be terminated by a `\0` and must not exceed 255 characters. Strings longer than 255 characters are truncated at the 255th character. If no message is desired, code the defined term **NULL**.

#### slist

This argument is a pointer to an array of pointers to type `void`, indicating extra areas of storage that are to be displayed on the dump. See *TPF General Macros* for a detailed explanation of **slist** format and effects. If no storage list exists, code this parameter as **NULL**.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example forces a storage dump bearing ID number U012345 (U is the default prefix) to be issued, with control returning to the program following the dump. The message, ERROR OCCURRED, is displayed at the prime CRAS and will be appended to the dump.

```
#include <tpfapi.h>
:
:
serrc_op(SERRC_RETURN,0x12345,"ERROR OCCURRED",NULL);
```

## Related Information

- “abort—Terminate Program Abnormally” on page 7
- “serrc\_op\_ext—Issue System Error Extended: Operational” on page 456
- “snapc—Issue Snapshot Dump” on page 508
- *TPF General Macros*, SERRC macro.

## serrc\_op\_ext—Issue System Error Extended: Operational

This function causes a storage dump to be issued by the system error routine, and may optionally display a message at prime CRAS. The ECB may be exited if desired. The `serrc_op_ext` function allows a prefix character to be concatenated with the system error number; therefore, you can distinguish between IBM calls and user calls that code the same system error number.

### Format

```
#include <tpfapi.h>
void serrc_op_ext(enum t_serrc status, int number, const char *msg,
                  char prefix, struct serrc_list **list);
```

#### status

The status of the ECB following the dump. This argument must belong to enumeration type `t_serrc`, which is defined in `tpfapi.h`. Code the defined term **SERRC\_EXIT** to force the ECB to exit, **SERRC\_RETURN** to cause a return to the calling program, or **SERRC\_CATA** to cause a catastrophic error.

#### number

The identification number for the dump. This argument is an integer and should be a unique number ranging from 1 to `X'FFFFFF'`.

#### msg

This argument is a pointer to type `char`, which is a message text string to be displayed at the CRAS console and appended to the dump. This string must be terminated by a `\0` and must not exceed 255 characters. Strings longer than 255 characters are truncated at the 255th character. If no message is desired, code the defined term **NULL**.

#### prefix

This argument is an uppercase alphabetic character that is concatenated with the system error number in the console message and in the dump. You can use any uppercase alphabetic characters in the following ranges: A-H and J-V. The letters I and W—Z are reserved for IBM use. If a **NULL** prefix is given then the prefix will be U.

#### list

This argument is a pointer to a pointer of type `struct serrc_list`, indicating areas of storage to be displayed in the dump. The `serrc_list` structure has the following fields:

##### serrc\_len

Type `short int`, the length of the area to be dumped. Code zero for `serrc_len` to indicate no more areas are to be dumped.

##### serrc\_name

Character string, the name of the data area. `serrc_name` must be 8 characters long, left justified, and padded with blanks.

##### serrc\_tag

The location of the area to be dumped.

##### serrc\_indir

Whether the address in `serrc_tag` is an indirection. To indicate an indirection, code the defined term **SERRC\_INDIR**. If `serrc_tag` is not an indirection, code the defined term **SERRC\_NOINDIR**.

If no storage list exists, code the defined term **NULL**.

**Note:** There is a limit of 50 entries in the serrc\_list pointer.

## Normal Return

Void.

## Error Return

Not applicable.

## Programming Considerations

The serrc\_list structure may only contain a maximum of 50 list elements. If this limit is exceeded a system error will be issued and the ECB will be exited.

## Examples

The following example forces a storage dump bearing ID number A012345 (A is the prefix) to be issued. The control returns to the program following the dump. The message, ERROR OCCURRED, is displayed at the prime CRAS and is appended to the dump.

```
#include <tpfapi.h>
:
serrc_op_ext(SERRC_RETURN,0x12345,"ERROR OCCURRED",'A',NULL);
```

## Related Information

- “abort—Terminate Program Abnormally” on page 7
- “serrc\_op—Issue System Error: Operational” on page 454
- “snapc—Issue Snapshot Dump” on page 508
- *TPF General Macros*, SERRC macro.

## serrc\_op\_slt–Issue System Error SLIST: Operational

This function causes a storage dump to be issued by the system error routine, and may optionally display a message at prime CRAS. The ECB may be exited if desired. The `serrc_op_ext` function allows a prefix character to be concatenated with the system error number; thus you can distinguish between IBM and user calls that code the same system error number. This function is designed to allow users who have coded the `serrc_op` function with the **slist** parameter to define their own prefixes with minimal change to the existing code. Other users should code the `serrc_op_ext` function.

### Format

```
#include <tpfapi.h>
void serrc_op_slt(enum t_serrc status, int number, const char *msg,
                 void *slist[], char prefix);
```

#### status

The status of the ECB following the dump. This argument must belong to enumeration type `t_serrc`, which is defined in `tpfapi.h`. Code the defined term **SERRC\_EXIT** to force the ECB to exit, **SERRC\_RETURN** to cause a return to the calling program, or **SERRC\_CATA** to cause a catastrophic error.

#### number

The identification number for the dump. This argument is an integer and should be a unique number ranging from 1 to X'FFFFFF'.

#### msg

This argument is a pointer to type `char`, which is a message text string to be displayed at the CRAS console and appended to the dump. This string must be terminated by a `\0` and must not exceed 255 characters. Strings longer than 255 characters are truncated at the 255th character. If no message is desired, code the defined term **NULL**.

#### slist

This argument is a pointer to an array of pointers to type `void`, indicating extra areas of storage which are to be displayed on the dump. See *TPF General Macros* for a detailed explanation of **slist** format and effects. If no storage list exists, code this parameter as **NULL**. If no storage list exists, code the defined term **NULL**.

#### prefix

This argument is an uppercase alphabetic character that is concatenated with the system error number in the console message and in the dump. You can use any uppercase alphabetic characters in the following ranges: A–H and J–V. The letters I and W–Z are reserved for IBM use. The default prefix is U.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

None.



## Examples

The following example forces a storage dump bearing ID number A012345 (A is the prefix) to be issued. The control returns to the program following the dump. The message, "ERROR OCCURRED", is displayed at the prime CRAS and is appended to the dump.

```
#include <tpfapi.h>
:
serrc_op_slb(SERRC_RETURN,0x12345,"ERROR OCCURRED", NULL,'A');
```

## Related Information

- "abort—Terminate Program Abnormally" on page 7
- "serrc\_op—Issue System Error: Operational" on page 454
- "serrc\_op\_ext—Issue System Error Extended: Operational" on page 456
- "snapc—Issue Snapshot Dump" on page 508
- *TPF General Macros*, SERRC macro.

---

## setbuf—Control Buffering

This function controls buffering.

### Format

```
#include <stdio.h>
void setbuf(FILE *stream, char *buffer);
```

**stream**

The newly opened file.

**buffer**

The user-supplied buffer.

This function controls the way data is buffered for the specified **stream**. The **stream** pointer must refer to an open file and the setbuf function must be the first operation on the stream.

If the **buffer** argument is NULL, the **stream** is unbuffered. If not, the stream will be full buffered and the **buffer** must point to an array length of at least BUFSIZ characters defined in the stdio.h header file. Input/Output (I/O) functions may use the **buffer**, which you specify here, for input/output buffering instead of the default system-allocated buffer for the given **stream**.

**Note:** If you use the setbuf or setvbuf function to define your own buffer for a stream, you must ensure that the buffer is available the whole time that the stream associated with the buffer is in use.

For example, if the buffer is an automatic array (block scope) and is associated with stream **s**, leaving the block causes the storage to be deallocated. I/O operations of stream **s** are prevented from using deallocated storage. Any operation on **s** would fail because the operation would attempt to access the nonexistent storage.

To ensure that the buffer is available throughout the life of a program, make the buffer a variable allocated at file scope. This can be done by using an identifier of type **array** declared at file scope or by allocating storage (with malloc or calloc) and assigning the storage address to a pointer declared at file scope.

### Normal Return

There is no returned value.

### Error Return

Not applicable.

### Programming Considerations

Your buffer will only be as large as BUFSIZ. If you define the **buffer** to be larger than BUFSIZ, your buffer will be limited to BUFSIZ.

### Examples

The following example opens the myfile.dat file for writing. It then calls the setbuf function to establish a buffer of length bufsiz.

```
#include <stdio.h>

int main(void)
{
```

```

char buf[BUFSIZ];
char string[] = "hello world";
FILE *stream;

stream = fopen("myfile.dat", "wb");
setbuf(stream,buf);          /* set up buffer */

fwrite(string, sizeof(string), 1, stream);

fclose(stream);
}
/* Note that the stream is closed before the buffer goes out */
/* of scope at the end of this block.                          */

```

## Related Information

- “fclose—Close File Stream” on page 127
- “fflush—Write Buffer to File” on page 144
- “fopen—Open a File” on page 199
- “setvbuf—Control Buffering” on page 474.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## setegid—Set the Effective Group ID

This function sets the effective group ID (GID) of the calling function's process to **gid**.

### Format

```
#include <unistd.h>
int setegid(uid_t gid);
```

**gid**

The GID numeric value used in setting the effective GID.

### Normal Return

If successful, the setegid function returns an integer value of 0.

### Error Return

If unsuccessful, the setegid function returns -1 and sets errno to one of the following:

#### EINVAL

The value specified for **gid** is incorrect.

#### EPERM

The process does not have the appropriate privileges or **gid** does not match the real GID or the saved set GID.

## Programming Considerations

This function sets the effective GID of the calling function's process to **gid** if **gid** is equal to the real GID or saved set GID of the calling process, or if the process has appropriate privileges.

The appropriate privileges are if the calling program has the key 0 option set on in the PAT or having the effective UID 0 (root). Otherwise, the real GID and the saved set GID are not changed.

## Examples

The following example provides information for the effective group ID (GID) of the caller and sets its effective GID:

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

int main(void) {

    printf("Your effective group id is %d\n", (int) getegid() );
    if ( setegid ( 100 ) != 0 )
        perror ( "setegid() error" );
    else
        printf("Your effective group id was changed to %d\n",
               (int) setegid() );
    return 0;
}
```

## Related Information

- “getegid—Get the Effective Group ID” on page 254
- “getgid—Get the Real Group ID” on page 260
- “setgid—Set the Group ID to a Specified Value” on page 468.

## setenv—Add, Change, or Delete an Environment Variable

This function adds, changes, or deletes environment variables.

### Format

```
#define _POSIX_SOURCE
#include <stdlib.h>
int setenv(const char *var_name, const char *new_value, int change_flag)
```

#### **var\_name**

A pointer to a character string that contains the name of the environment variable to be added, changed, or deleted.

#### **new\_value**

A pointer to a character string that contains the value of the environment variable named in **var\_name**.

#### **change\_flag**

A flag that can take any integer value:

##### **Nonzero**

Change the existing entry.

If **var\_name** is already defined and exists in the environment list, change the existing entry to **new\_value**.

If **var\_name** was previously undefined, it is added to the environment list.

##### **0** Do *not* change the existing entry.

If **var\_name** is defined and exists in the environment list, the value is *not* changed to **new\_value**.

If **var\_name** was previously undefined, it is added to the environment list.

### Normal Return

0

### Error Return

−1 and sets `errno` to one of the following:

**EINVAL** **var\_name** is a null pointer that points to either an empty string or to a string that contains an equal sign (=).

**ENOMEM** There is not enough storage available to the program to extend the environment list.

### Programming Considerations

- Only the private environment list of the ECB is changed; the `setenv` function does not change the global default environment list.
- Environment variables set with the `setenv` function only exist for the life of the ECB and are not saved when the ECB ends.
- If the `setenv` function is called with **var\_name** containing an equal sign (=), `setenv` fails and `errno` is set to indicate that an incorrect argument was passed to the function.

### Examples

The following example sets up and prints the value of the environment variable `_EDC_ANSI_OPEN_DEFAULT`.

## setenv

```
#include <stdio.h>
#define _POSIX_SOURCE
#include <stdlib.h>

int main(void)
{
    char *x;

    /* set environment variable _EDC_ANSI_OPEN_DEFAULT to "Y" */
    setenv("_EDC_ANSI_OPEN_DEFAULT", "Y", 1);

    /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT */
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("program1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");
    /* call the child program */
    system("program2");

    /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT */
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("program1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");
}
```

### Output

```
program1 _EDC_ANSI_OPEN_DEFAULT = Y
program1 _EDC_ANSI_OPEN_DEFAULT = Y
```

**Y** The value defined for the environment variable.

In the following example, a child ECB of the previous example was started by a system call. Like the previous example, this example sets up and prints the value of the environment variable `_EDC_ANSI_OPEN_DEFAULT`. The program then deletes the environment value and prints the value as undefined. This example shows that environment variables are propagated forward to a child ECB, but not backward to the parent ECB.

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

int main(void)
{
    char *x;

    /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT */
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("program2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");

    /* clear the Environment Variables Table */
    unsetenv("_EDC_ANSI_OPEN_DEFAULT");

    /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT */
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("program2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");
}
```

### Output

```
program2 _EDC_ANSI_OPEN_DEFAULT = Y
program2 _EDC_ANSI_OPEN_DEFAULT = undefined
```

**Y** The value defined for the environment variable.

## **Related Information**

- “getenv—Get Value of Environment Variables” on page 255
- “system—Execute a Command” on page 527
- “unsetenv—Delete an Environment Variable” on page 671.

## seteuid—Set the Effective User ID

This function sets the effective user ID (UID) to **uid**.

### Format

```
#include <unistd.h>
int seteuid(uid_t uid);
```

**uid**

The user ID numeric value used in setting the effective user ID.

### Normal Return

If successful, the seteuid function returns an integer value of 0 and the effective user ID is set to **uid**. The real UID and the saved set UID are not changed.

### Error Return

If unsuccessful, the seteuid function returns -1 and sets errno to one of the following:

#### EINVAL

The value specified for **uid** is not supported.

#### EPERM

The process does not have the appropriate privileges or **uid** does not match the real UID or the saved set UID.

- The calling program does not have the same real UID or the same saved set UID.
- The calling program is not a root user ID or superuser.
- The calling program does not have the key 0 macro option set on in the program allocation table (PAT).

## Programming Considerations

- This function is successful when **uid** is equal to the real **uid** or the saved set-user-ID of the calling process, or if the process has appropriate privileges.

The appropriate privileges are when the calling program has the key 0 macro option set on in the PAT or when the process is a superuser.

## Examples

The following example provides information for the effective user ID of the caller and sets its effective UID.

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

int main(void) {

    printf("Your effective user id is %d\n", (int) geteuid() );
    if ( seteuid ( 25 ) != 0 )
        perror ( "seteuid() error" );
    else
        printf("Your effective user id was changed to %d\n",
               (int) geteuid() );
    return 0;
}
```

The seteuid function returns 0.



## **Related Information**

- “geteuid–Get the Effective User ID” on page 256
- “getpwuid–Access the User Database by User ID” on page 271
- “getuid–Get the Real User ID” on page 276
- “setuid–Set the Real User ID” on page 472.

---

## setgid–Set the Group ID to a Specified Value

This function sets one or more of the group IDs (GIDs) for the current process to a specified value.

### Format

```
#include <unistd.h>
int setgid(uid_t gid);
```

**gid**

The intended new group ID value.

### Normal Return

If successful, the setgid function returns an integer value of 0 and the group ID is set.

### Error Return

If unsuccessful, the setgid function returns –1 and sets errno to one of the following:

**EINVAL**

The value specified for **gid** is incorrect.

**EPERM**

The process does not have one of the appropriate privileges to set **gid**.

- The calling program process is not the same real group ID as **gid** or the same saved set-group-ID as **gid**.
- The calling program process is not a root user or superuser.
- The calling program does not have the key 0 macro option set on in the program allocation table (PAT).

## Programming Considerations

- If the process has appropriate privileges, the setgid function sets the real group ID, effective group ID, and the saved set-group-ID to **gid**.
- If the process does not have appropriate privileges, but the group ID is equal to the real group ID or the saved set-group-ID, the setgid function sets the effective group ID to **gid**; the real group ID and saved set group ID remain unchanged.

The appropriate privileges are when the calling program has the key 0 option set on in the PAT or when the effective UID is equal to superuser.

## Examples

The following example changes the effective group ID.

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

int main(void) {

    printf("Your group id is %d\n", (int) getgid() );
    if ( setgid ( 100 ) != 0 )
        perror ( "setgid() error" );
    else
        printf("Your group id was changed to %d\n",
               (int) getgid() );
    return 0;
}
```

## **Related Information**

- “getegid–Get the Effective Group ID” on page 254
- “getgid–Get the Real Group ID” on page 260
- “setgid–Set the Group ID to a Specified Value” on page 468.

## setjmp–Preserve Stack Environment

This function saves a stack environment that can then be restored by the `longjmp` function. The nesting level is also saved to validate a `longjmp` call. The `setjmp` and `longjmp` functions provide a way to perform a nonlocal goto. They are often used in signal handlers.

A call to the `setjmp` function causes it to save the current stack environment in **env**. The next call to the `longjmp` function restores the saved environment and returns control to a point corresponding to the `setjmp` call. The `setjmp` and `longjmp` functions are not restricted to the same dynamic load module (DLM).

### Format

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

#### env

The `jmp_buf` for saving the current stack and other environment information.

### Normal Return

Returns the value 0 after saving the stack environment. If the `setjmp` function returns as a result of a `longjmp` call, it returns the **value** argument of `longjmp`, or the value 1 if the **value** argument of the `longjmp` function is equal to 0.

### Error Return

None.

## Programming Considerations

- Ensure that the function that calls the `setjmp` function does not return before you call the corresponding `longjmp` function. Calling the `longjmp` function after the function calling `setjmp` returns causes unpredictable program behavior.
- The database ID (DBI), program base ID (PBI), subsystem user (SSU) ID, or any other application program global pointer that might have changed after the `setjmp` function is issued needs to be restored by users prior to calling the `longjmp` function.
- The values of all variables, except register variables and nonvolatile automatic variables that are accessible to the function receiving control, contain the values they had when the `longjmp` function was called.
- The values of register variables are unpredictable.
- Nonvolatile *auto* variables that are changed between calls to the `setjmp` and `longjmp` functions are also unpredictable.
- You can call the `setjmp` function in one of the following contexts only:
  - The entire controlling expression of a selection or iteration statement
  - One operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement
  - The operand of a unary “!” operator with the resulting expression being the entire controlling expression of a selection or iteration
  - The entire expression of an expression statement (possibly cast to void).

## Examples

This example stores the stack environment at the statement:

```
if(setjmp(mark) != 0) ...
```

When the program first performs the `if` statement, it saves the environment in `mark` and sets the condition to FALSE because `setjmp` returns the value 0 when it saves the environment. The program prints the following message: `setjmp has been called`.

The next call to function `p` tests for a local error condition, which can cause it to perform the `longjmp` function. (The function `p` that performs the `setjmp` function can be in the same DLM or a separate DLM). Then control returns to the original `setjmp` function using the environment saved in `mark`. This time the condition is TRUE because `-1` is the returned value from the `longjmp` function. The program then performs the statements in the block and prints the following: `longjmp has been called`. Finally, the program calls the `recover` function and returns.

```
/* This example shows the effect of having set the stack environment. */
#include <stdio.h>
#include <setjmp.h>
jmp_buf mark;
void p(void);
void recover(void);
int ADLM(void)
{
    if (setjmp(mark) != 0) {
        printf("longjmp has been called\n");
        recover();
        return(1);
    }
    printf("setjmp has been called\n");
    :
    p();
    :
    return(32);      /* Return suitable value */
}
void p(void)
{
    int error = 0;
    :
    error = 9;
    :
    if (error != 0)
        longjmp(mark, -1);
    :
}
void recover(void)
{
    :
}
```

## Related Information

“`longjmp`—Restore Stack Environment” on page 313.

---

## setuid–Set the Real User ID

This function sets the real, effective, or saved set-user-IDs (UIDs) for the current process to **uid**.

### Format

```
#include <unistd.h>
int setuid(uid_t uid);
```

**uid**

The intended new UID value.

### Normal Return

If successful, the setuid function returns 0 and sets the real, effective, or saved set-user-IDs for the current process to **uid**.

### Error Return

If unsuccessful, the setuid function returns –1 and sets errno to one of the following:

#### EINVAL

The value of **uid** is not valid.

#### EPERM

The process does not have appropriate privileges and **uid** does not match the real user ID or the saved set-user-ID.

- The calling program process is not the same real user ID or the same set-user-ID as **uid**.
- The calling program process is not a root user ID or superuser.
- The calling program does not have the key 0 macro option set on in the program allocation table (PAT).

### Programming Considerations

- If the process has appropriate privileges, the setuid function sets the real UID, effective UID, and saved-set-UID to **uid**.
- If the process does not have appropriate privileges, but **uid** is equal to the real user ID or the saved set-user-ID, the setuid function sets the effective user ID to **uid**; the real user ID and saved set user ID remain unchanged.

The appropriate privileges are when the calling program has the key 0 macro option set on in the PAT or when the process is superuser.

### Examples

The following example changes the real, effective, and saved-set-UIDs.

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

int main(void) {
    printf ( "prior to setuid(), uid = %d, effective uid = %d\n"
            (int) getuid(), (int) geteuid() );
    if ( setuid(25) != 0 )
        perror( "setuid() error" );
    else
```

```
        printf ( "after setuid(),   uid = %d, effective uid = %d\n",  
                (int) getuid(), (int) geteuid() );  
    return 0;  
}
```

## Related Information

- “getuid—Get the Real User ID” on page 276
- “seteuid—Set the Effective User ID” on page 466.

## setvbuf—Control Buffering

This function controls the buffering strategy and buffer size for a specified stream.

### Format

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

#### **stream**

The newly opened file.

#### **buf**

The user-supplied buffer.

#### **type**

The type of buffering for **stream**.

#### **size**

The size of the user-supplied buffer.

The **stream** pointer must refer to an open file and the setvbuf function must be the first operation on the file.

The setvbuf function is more flexible than the setbuf function because you can specify the type of buffering and the size of the buffer.

The location pointed to by **buf** designates an area that you provide that the C run-time library can choose to use as a buffer for the stream. A **buf** value of NULL indicates that no such area is supplied and that the C run-time library is to assume responsibility for managing its own buffers for the stream. If you supply a buffer, it must exist until the stream is closed.

If **type** is `_IOFBF` or `_IOLBF`, **size** is the size of the supplied buffer. If **buf** is NULL, the C library will take **size** as the suggested size for its own buffer. If **type** is `_IONBF`, both **buf** and **size** are ignored.

Value	Meaning
<code>_IONBF</code>	No buffer is used.
<code>_IOFBF</code>	Full buffering is used for input and output. Use <b>buf</b> as the buffer and <b>size</b> as the size of the buffer.
<code>_IOLBF</code>	Line buffering is used for text stream I/O. The buffer is flushed when a new-line character is used (text stream) or when the buffer is full.

The value for **size** must be greater than 0.

**Note:** If you use the setvbuf or setbuf function to define your own buffer for a stream, you must ensure that either the buffer is available after the program ends or the stream is closed or flushed before you call `exit`. This can be done by defining the array with a file scope or by dynamically allocating the storage for the array using `malloc`.

For example, if the buffer is declared the scope of a function block, the **stream** must be closed before the function ends. This prevents the storage allocated to the buffer from being used after it has been deallocated.



## Normal Return

If successful or if this function chooses not to use your buffer, setvbuf returns a value of 0.

## Error Return

The setvbuf function returns a nonzero value if an incorrect value was specified in the parameter list or if the request cannot be performed.

## Programming Considerations

None.

## Examples

The following example sets up a buffer of buf for stream1 and specifies that input from stream2 is to be unbuffered.

```
#include <stdio.h>
#define BUF_SIZE 256

char buf[BUF_SIZE];

int main(void)
{
    FILE *stream1 = fopen("myfile1.dat", "r");
    FILE *stream2 = fopen("myfile2.dat", "r");

    /* stream1 uses line buffering */
    if (setvbuf(stream1, buf, _IOLBF, sizeof(buf)) != 0)
        printf("Incorrect type or size of buffer 1");

    /* stream2 is unbuffered */
    if (setvbuf(stream2, NULL, _IONBF, 0) != 0)
        printf("Incorrect type or size of buffer 2");
    :
}
```

## Related Information

- “fclose—Close File Stream” on page 127
- “fflush—Write Buffer to File” on page 144
- “fopen—Open a File” on page 199
- “setbuf—Control Buffering” on page 460.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## shmat—Attach Shared Memory

This function attaches the shared memory associated with the specified shared memory identifier to the address space of the calling process.

### Format

```
#include <sys/shm.h>
void *shmat(int shm_id,
            const void *shmaddr,
            int shmflg);
```

#### shm\_id

The shared memory identifier associated with the shared memory to be attached. The shared memory identifier is returned by the `shmget` function.

#### shmaddr

Reserved for future IBM use. This must be set to `NULL`.

#### shmflg

Reserved for future IBM use. This must be set to 0.

#### TPF deviation from POSIX

This parameter is provided for compatibility with a UNIX environment. The TPF system does not use this parameter.

### Normal Return

If successful, the `shmat` function returns the address of the shared memory and the `shm_nattch` field in the `shm_id_ds` data structure is incremented by 1.

### Error Return

If unsuccessful, the `shmat` function returns a value of `-1` and sets `errno` to one of the following:

<b>EACCES</b>	Operation permission is denied to the calling process.
<b>EINVAL</b>	The value of the <b>shm_id</b> parameter is not a valid shared memory identifier or the <b>shmaddr</b> parameter is not set to <code>NULL</code> .
<b>EMFILE</b>	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

### Programming Considerations

The `shm_id_ds` data structure is defined in the `sys/shm.h` header file.

### Examples

The following example attaches shared memory to the address space of the calling process by using the shared memory identifier returned by the `shmget` function. The shared memory identifier is removed from the TPF system by using the `shmctl` function when it is no longer needed.

```
#include <sys/ipc.h>
#include <sys/shm.h>

void main(void)
{
    key_t i;
    int shm;
```

```
void *addr;
struct shmid_ds buf;

i = ftok("/usr",3);
shm = shmget(i,8000,IPC_CREAT+S_IRUSR+S_IWUSR);
addr = shmat(shm,NULL,0);

i = shmctl(shm,IPC_RMID,&buf);

}
```

## Related Information

- “ftok—Generate a Token” on page 235
- “shmctl—Shared Memory Control” on page 478
- “shmdt—Detach Shared Memory” on page 481
- “shmget—Allocate Shared Memory” on page 483.

## shmctl–Shared Memory Control

This function provides a variety of operations that are used to control the shared memory identified by the **shmid** parameter.

### Format

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shm_ds *buf);
```

#### shmid

The shared memory identifier for which this operation is to take place. The shared memory identifier is returned by the `shmget` function.

#### cmd

The shared memory control operation to be performed, which is specified as one of the following:

##### IPC\_STAT

Use this operation to obtain status information for the shared memory identified by the **shmid** parameter. The current value of each field in the `shm_ds` data structure associated with the **shmid** parameter is placed in the data structure pointed to by the **buf** parameter. The contents of this structure are defined in the `sys/shm.h` header file. This operation requires read permission for the shared memory identified by the **shmid** parameter.

##### IPC\_SET

Use this operation to set the values of the following fields of the `shm_ds` data structure associated with the **shmid** parameter to the corresponding value in the structure pointed to by the **buf** parameter:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode (only the low-order 9 bits)
```

This operation can be performed only by a process that has an effective user ID (UID) equal to one of the following:

- A process with the same privileges as the shared memory identified by the **shmid** parameter
- The value of `shm_perm.cuid` or `shm_perm.uid` in the `shm_ds` data structure associated with the **shmid** parameter.

##### IPC\_RMID

Use this operation to remove the following from the TPF system:

- The shared memory identifier specified by the **shmid** parameter
- The shared memory associated with the **shmid** parameter
- The `shm_ds` data structure associated with the **shmid** parameter.

This operation can be performed only by a process that has an effective UID equal to one of the following:

- A process with the same privileges as the shared memory identified by the **shmid** parameter
- The value of `shm_perm.cuid` or `shm_perm.uid` in the `shm_ds` data structure associated with the **shmid** parameter.

Removal is completed asynchronously to the return from the `shmctl` function when the last attached shared memory segment is detached. When `IPC_RMID` is processed, no more attaches for the shared memory identified by the **shmid** parameter are allowed.

**buf**

A pointer to a `shmid_ds` data structure as defined in the `sys/shm.h` header file.

## Normal Return

If successful, the `shmctl` function returns a value of 0.

## Error Return

If unsuccessful, the `shmctl` function returns a value of `-1` and sets `errno` to one of the following:

- |                |  |
|----------------|--|
| <b>EACCESS</b> | The <b>cmd</b> parameter is specified as <code>IPC_STAT</code> , but the calling process does not have read permission for the shared memory.  |
| <b>EINVAL</b>  | The value of the <b>shmid</b> parameter is not a valid shared memory identifier or the value of the <b>cmd</b> parameter is not a valid operation.   |
| <b>EPERM</b>   | <p>The <b>cmd</b> parameter is specified as either <code>IPC_RMID</code> or <code>IPC_SET</code> and one of the following is true:</p> <ul style="list-style-type: none"> <li>• The effective UID of the calling process is not equal to that of a process with the same access privileges as for the shared memory.</li> <li>• The effective UID of the calling process is not equal to the value of <code>shm_perm.cuid</code> or <code>shm_perm.uid</code> in the data structure associated with the <b>shmid</b> parameter.</li> </ul> |

## Programming Considerations

- Programs require restricted authorization to use this function.
- When `IPC_STAT` is specified for the **cmd** parameter, the **buf** parameter is a pointer to a `shmid_ds` data structure where the current values of each member of the `shmid_ds` data structure associated with the **shmid** parameter are returned to the caller.
- When `IPC_SET` is specified for the **cmd** parameter, the **buf** parameter is a pointer to a `shmid_ds` data structure where the caller specifies values for `shm_perm.uid`, `shm_perm.gid`, and `shm_perm.mode`.

## Examples

The following example attaches shared memory to the address space of the calling process by using the shared memory identifier returned by the `shmget` function. The shared memory identifier is removed from the TPF system by using the `shmctl` function when it is no longer needed.

```
#include <sys/ipc.h>
#include <sys/shm.h>

void main(void)
{
    key_t i;
    int shm;
    void *addr;
    struct shmid_ds buf;

    i = ftok("/usr",3);
    shm = shmget(i,8000,IPC_CREAT+S_IRUSR+S_IWUSR);
```

## shmctl

```
addr = shmat(shm, NULL, 0);  
  
i = shmctl(shm, IPC_RMID, &buf);  
  
}
```

## Related Information

- “ftok—Generate a Token” on page 235
- “shmat—Attach Shared Memory” on page 476
- “shmdt—Detach Shared Memory” on page 481
- “shmget—Allocate Shared Memory” on page 483.

## shmdt–Detach Shared Memory

This function detaches the shared memory segment located at the specified address from the address space of the calling process.

### Format

```
#include <sys/shm.h>
type      shmdt(const void *shmaddr);
```

#### shmaddr

The address of the shared memory segment to be detached from the address space of the calling process. This address is returned by the shmat function.

### Normal Return

If successful, the shmdt function returns a value of 0 and the value of shm\_nattach in the shmid\_ds data structure is decremented by 1.

### Error Return

If unsuccessful, the shmdt function returns a value of –1 and sets errno to the following:

#### EINVAL

The value of the **shmaddr** parameter is not the address of the first byte of the shared memory.

## Programming Considerations

- The shmid\_ds data structure is defined in the sys/shm.h header file.
- On return, the process should not write to this storage because it may or may not still be viewable.
- Programs require restricted authorization to use this function.

## Examples

The following example attaches shared memory to the address space of the calling process by using the shared memory identifier returned by the shmget function. The shared memory identifier is removed from the TPF system by using the shmctl function when it is no longer needed.

```
#include <sys/ipc.h>
#include <sys/shm.h>
int main()
{
    cpmst *addr;
    key_t i;
    int shm;

    /** addr is gotten via a shmat **/
    i = ftok("/usr",3);
    shm = shmget(i,8000,IPC_CREAT+S_IRUSR+S_IWUSR);
    addr = shmat(shm,NULL,0);

    i = shmdt(addr);
}
```

**shmdt**

## **Related Information**

- “ftok—Generate a Token” on page 235
- “shmat—Attach Shared Memory” on page 476
- “shmctl—Shared Memory Control” on page 478
- “shmget—Allocate Shared Memory” on page 483.



## shmget—Allocate Shared Memory

This function allocates shared memory for the specified key and returns its associated shared memory identifier. The shared memory identifier is used as the **shmid** parameter for the shared memory attach (`shmat`) and shared memory control (`shmctl`) functions.

### Format

```
#include <sys/shm.h>
int shmget( key_t key,
            size_t size,
            int shmflg);
```

#### key

One of the following:

- A key returned by the `ftok` function if the shared memory can be accessed by more than one process.
- The value `IPC_PRIVATE` to indicate that the shared memory cannot be accessed by other processes.

A shared memory identifier, associated data structure, and the shared memory are created for the **key** parameter if one of the following is true:

- The **key** parameter has a value of `IPC_PRIVATE`.
- The **key** parameter does not already have a shared memory identifier associated with it and `IPC_CREAT` is specified for the **shmflg** parameter.

#### size

The size of the shared memory to be created, in bytes.

#### shmflg

A flag field specified with any combination of the following constants defined in the `sys/shm.h` and `modes.h` header files:

##### IPC\_CREAT

Create a shared memory segment if a shared memory identifier does not exist for the specified **key** parameter.

`IPC_CREAT` is ignored when `IPC_PRIVATE` is specified for the **key** parameter.

##### IPC\_EXCL

Causes the `shmget` function to fail if the specified **key** parameter has an associated shared memory identifier.

`IPC_EXCL` is ignored when `IPC_CREAT` is not specified for the **shmflg** parameter or `IPC_PRIVATE` is specified for the **key** parameter.

##### S\_IRUSR

Permits read access when the effective user ID (UID) of the caller is equal to either the `shm_perm.cuid` or the `shm_perm.uid` field in the `shmid_ds` data structure associated with the shared memory identified by the **key** parameter.

##### S\_IWUSR

Permits write access when the effective UID of the caller is equal to either the `shm_perm.cuid` or the `shm_perm.uid` field in the `shmid_ds` data structure associated with the shared memory identified by the **key** parameter.

##### S\_IRGRP

Permits read access when the effective group ID (GID) of the caller is equal

## shmget

to either the `shm_perm.cgid` or the `shm_perm.gid` field in the `shmid_ds` data structure associated with the shared memory identified by the **key** parameter.

### S\_IWGRP

Permits write access when the effective GID of the caller is equal to either the `shm_perm.cgid` or the `shm_perm.gid` field in the `shmid_ds` data structure associated with the shared memory identified by the **key** parameter.

### S\_IROTH

Permits other read access to the shared memory.

### S\_IWOTH

Permits other write access to the shared memory.

## Normal Return

If successful, the `shmget` function returns the shared memory identifier associated with the specified key as a nonnegative integer and a `shmid_ds` data structure is created for the allocated shared memory.

## Error Return

If unsuccessful, the `shmget` function returns a value of `-1` and sets `errno` to one of the following:

<b>EACCES</b>	A shared memory identifier exists for the <b>key</b> parameter, but operation permission as specified by the low-order 9 bits of the <b>shmflg</b> parameter could not be granted because the UID of the calling process does not have the appropriate privileges to access the shared memory.
<b>EEXIST</b>	A shared memory identifier exists for the <b>key</b> parameter and both <code>IPC_CREAT</code> and <code>IPC_EXCL</code> are specified for the <b>shmflg</b> parameter.
<b>EINVAL</b>	One of the following conditions occurred: <ul style="list-style-type: none"><li>• A shared memory identifier does not exist for the specified <b>key</b> parameter and the value of the <b>size</b> parameter is less than the minimum size or greater than the maximum size imposed by the TPF system.</li><li>• A shared memory identifier exists for the specified <b>key</b> parameter, but the size of the shared memory is less than what is specified by the <b>size</b> parameter.</li><li>• A value of 0 was specified for the <b>shmflg</b> parameter.</li></ul>
<b>ENOENT</b>	A shared memory identifier does not exist for the <b>key</b> parameter and <code>IPC_CREAT</code> is not specified for the <b>shmflg</b> parameter.
<b>ENOMEM</b>	A shared memory identifier and associated shared memory segment are to be created, but there is not enough system storage available to satisfy the request.
<b>ENOSPC</b>	A shared memory identifier is to be created, but the limit (imposed by the TPF system) on the maximum number of shared memory identifiers that are allocated systemwide would be exceeded.

## Programming Considerations

- Programs require restricted authorization to use this function.
- The `shmid_ds` data structure is defined in the `sys/shm.h` header file.

- The following fields are initialized when a `shmid_ds` data structure is created for the allocated shared memory:
  - The `shm_perm.cuid` and `shm_perm.uid` fields are set to the effective UID of the calling process.
  - The `shm_perm.cgid` and `shm_perm.gid` fields are set to the effective GID of the calling process.
  - The low-order 9 bits of the `shm_perm.mode` field are set to the value in the low-order 9 bits of the **shmflg** parameter.
  - The `shm_segsz` field is set to the value of the **size** parameter.
  - The `shm_lpid`, `shm_nattach`, `shm_atime`, and `shm_dtime` fields are set to zero.
  - The `shm_ctime` field is set to the current time.

## Examples

The following example allocates shared memory and returns its shared memory identifier after calling the `ftok` function to return a key that is used as input to the `shmget` function.

```
#include <sys/ipc.h>
#include <sys/shm.h>
int main()
{
    key_t key;
    int i;

    key = ftok("/usr/testfile",3);
    i = shmget(key,100,IPC_CREAT);
}
```

## Related Information

- “`ftok`—Generate a Token” on page 235
- “`shmat`—Attach Shared Memory” on page 476
- “`shmctl`—Shared Memory Control” on page 478
- “`shmdt`—Detach Shared Memory” on page 481.

## sigaction—Examine and Change Signal Action

This function allows the calling process to examine or change the action associated with a signal.

**Note:** The `sigaction` function is similar to the `signal` function.

### Format

```
#include <signal.h>
int sigaction(int sig,
              const struct sigaction *act,
              struct sigaction *oact);
```

#### sig

One of the signals defined in the `signal.h` header file that corresponds to a signal to be examined or changed. See Table 13 on page 495 for a list of supported signals.

#### act

One of the following:

- A pointer to a `sigaction` data structure that is initialized by the caller. The signal action is changed according to the `sigaction` data structure referenced by the **act** parameter.
- A value of `NULL`. The signal action is not changed.

#### oact

One of the following:

- A pointer to a `sigaction` data structure where the `sigaction` function stores information related to the signal identified by the **sig** parameter before changing the signal action.
- A value of `NULL`. This causes the `sigaction` function to not store information related to the signal identified by the **sig** parameter.

### Normal Return

If successful, the `sigaction` function returns a value of 0.

### Error Return

If unsuccessful, the `sigaction` function returns a value of `-1` and sets `errno` to following:

#### EINVAL

One of the following is true:

- The **sig** parameter is not a valid signal number.
- An attempt was made to set the action to `SIG_DFL` for a signal (specified by the **sig** parameter) that cannot be caught or ignored.

### Programming Considerations

- When a signal action is changed by either the `sigaction` or the `signal` function, the new signal action replaces the previous signal action regardless of whether the previous signal action was installed by a `sigaction` or `signal` function.
- When a signal handler that is installed by the `sigaction` function is given control, a new signal mask is created for the process. The new signal mask includes all of the signals in the current signal mask and all of the signals in the `sa_mask` field in the `sigaction` data structure specified by the **act** parameter. In addition, if `SA_NODEFER` and `SA_RESETHAND` are not set in the `sa_flags` field in the

sigaction data structure specified by the **act** parameter, the new signal mask also includes the signal being handled. When the signal handler returns control, the signal mask is restored to the mask that was in use before the signal handler received control.

If SA\_NODEFER or SA\_RESETHAND is set on in the `sa_flags` field in the sigaction data structure specified by the **act** parameter, the signal mask is not changed.

- When the **oact** parameter is a pointer, the action for the specified signal is stored in the sigaction data structure referenced by the **oact** parameter. Moreover, if the action for the specified signal was installed by a `signal` function instead of by a `sigaction` function, the information stored in the sigaction data structure referenced by the **oact** parameter by a subsequent `sigaction` function call cannot be predicted.
- Table 12 summarizes sigaction function processing based on the specifications for the **act** and **oact** parameters:

Table 12. *sigaction* Function Specification Summary

<b>act</b>	<b>oact</b>	<b>Results</b>
Pointer	NULL	The signal action is changed as specified by the sigaction data structure specified by the <b>act</b> parameter.
Pointer	Pointer	The following actions occur: <ul style="list-style-type: none"> <li>– The signal action is changed as specified by the sigaction data structure specified by the <b>act</b> parameter.</li> <li>– The signal action before sigaction function processing occurs is stored in the sigaction data structure specified by the <b>oact</b> parameter.</li> </ul>
NULL	NULL	No action.
NULL	Pointer	The signal action before sigaction function processing occurs is stored in the sigaction data structure specified by the <b>oact</b> parameter.

- The following fields are defined in the sigaction data structure:

#### **sa\_handler**

The action to be associated with the signal specified by the **sig** parameter. This parameter is specified as one of the following:

#### **SIG\_DFL**

Indicates that the default action defined for the signal specified by the **sig** parameter is to be taken if this signal is raised.

#### **SIG\_IGN**

Indicates that the signal specified by the **sig** parameter is to be ignored.

#### **A pointer**

To a function that is given control if the signal specified by the **sig** parameter is raised.

#### **sa\_mask**

A signal mask of type `sigset_t` that specifies an additional set of signals to be blocked during processing by the signal handling function specified by the `sa_handler` field.

#### **sa\_flags**

Flags that affect the behavior of the signal. Any combination of the following flags defined in the `signal.h` header file can be set on in the `sa_flags` field.

## sigaction

### SA\_RESETHAND

If set on, the disposition of the signal handler is automatically reset to SIG\_DFL, and the SA\_SIGINFO flag is set off at entry to the signal handler. Additionally, if the SA\_RESETHAND flag is set on, the sigaction function behaves as if the SA\_NODEFER flag is set on.

### SA\_SIGINFO

This flag controls how the signal handler specified by the sa\_handler field is entered.

If set off, the signal handler prototype is:

```
void func(int signo);
```

If set on, the signal handler prototype is:

```
void func(int signo, siginfo_t *info, void *context);
```

The **info** parameter in the prototype is a pointer to a siginfo\_t data structure, which is defined in the signal.h header file and which describes why the signal was generated. The **context** parameter in the prototype is always NULL.

### SA\_NOCLDWAIT

When the signal specified by the **sig** parameter is SIGCHLD and if the SA\_NOCLDWAIT flag is set on, exit status is not saved for the child processes of the calling process. However, a SIGCHLD signal is sent to the calling process when a child process exits.

Otherwise this flag is ignored.

### SA\_NODEFER

If set on when the signal specified by the **sig** parameter is raised, this signal is not added to the signal mask of the calling process on entry to the signal handler unless it is included in the sa\_mask field.

If set off when the signal specified by the **sig** parameter is raised, this signal is added to the signal mask of the calling process on entry to the signal handler.

## Examples

The following example establishes a signal handler by using the sigaction function.

```
#include <sysapi.h>
#include <signal.h>
...
{
void ChldHndlr(int, siginfo_t *, void *);

struct tpf_fork_input create_parameters;
struct sigaction      act, oact;

pid_t child_pid;

/* initialize the new signal action */
act.sa_handler = ChldHndlr;
sigempty

set(&act.sa_mask);
act.sa_flags = 0;

/* install SIG_CHLD signal handler */
sigaction(SIG_CHLD, &act, &oact);
```

```

/* processing loop */
:

/* set up create_parameters */
create_parameters.program = "/bin/usr/usr1/app1.exe"
:

child_pid = tpf_fork(&create_parameters);
:

/* end of processing loop */
return;
}
void ChldHndlr(int signo, siginfo_t *info, void *context) {
    int  chld_state;

    {
        /* If someone used kill() to send SIGCHLD, ignore it */
        if (info.si_si_code != SI_USER)
        {
            exited_pid = wait( &chld_state );
            /* query return status */

:

        }
        return;
    }
}

```

## Related Information

- “raise—Raise Condition” on page 406
- “signal—Install Signal Handler” on page 495
- “sigpending—Examine Pending Signals” on page 498
- “sigprocmask—Examine and Change Blocked Signals” on page 499
- “sigsuspend—Set Signal Mask and Wait for a Signal” on page 502.

---

## sigaddset—Add a Signal to a Signal Set

This macro adds a specific signal to a signal set. It is one of several macros that manage or query signal sets.

### Format

```
#include <signal.h>
int sigaddset(sigset_t *set,
              int signo);
```

**set**

A pointer to a signal set of the `sigset_t` type.

**signo**

One of the signals defined in the `signal.h` header file that corresponds to a specific signal to be added. See Table 13 on page 495 for a list of supported signals.

### Normal Return

If successful, the `sigaddset` macro returns a value of 0.

### Error Return

If an error occurs, the `sigaddset` macro returns a value of `-1` and sets `errno` to the following:

**EINVAL**

The value specified by the **signo** parameter is not a valid signal number.

## Programming Considerations

Before using the `sigaddset` macro to add a signal to a signal set, the signal set must be initialized by either the `sigemptyset` or the `sigfillset` macro. If it is not initialized, the contents of the signal set are not guaranteed.

## Examples

See “`sigsuspend`—Set Signal Mask and Wait for a Signal” on page 502 for an example of the `sigaddset` macro.

## Related Information

- “`sigaction`—Examine and Change Signal Action” on page 486
- “`sigdelset`—Delete a Signal from a Signal Set” on page 491
- “`sigemptyset`—Initialize and Empty a Signal Set” on page 492
- “`sigfillset`—Initialize and Fill a Signal Set” on page 493
- “`sigismember`—See If a Signal Is a Member of a Signal Set” on page 494
- “`signal`—Install Signal Handler” on page 495
- “`sigpending`—Examine Pending Signals” on page 498
- “`sigprocmask`—Examine and Change Blocked Signals” on page 499
- “`sigsuspend`—Set Signal Mask and Wait for a Signal” on page 502.



## sigdelset—Delete a Signal from a Signal Set

This macro deletes a specific signal from a signal set. It is one of several macros that manage or query signal sets.

### Format

```
#include <signal.h>
int sigdelset(sigset_t *set,
              int signo);
```

#### set

A pointer to a signal set of the `sigset_t` type.

#### signo

One of the signals defined in the `signal.h` header file that corresponds to a specific signal to be deleted. See Table 13 on page 495 for a list of supported signals.

### Normal Return

On successful completion, the `sigdelset` macro returns a value of 0.

### Error Return

If an error occurs, the `sigdelset` macro returns a value of `-1` and sets `errno` to the following:

#### EINVAL

The value specified by the **signo** parameter is not a valid signal number.

## Programming Considerations

Before using the `sigdelset` macro to delete a signal from a signal set, the signal set must be initialized by using either the `sigemptyset` macro or `sigfillset` macro. If it is not initialized, the contents of the signal set are not guaranteed.

## Examples

See “`sigsuspend—Set Signal Mask and Wait for a Signal`” on page 502 for an example of the `sigdelset` macro.

## Related Information

- “`sigaction—Examine and Change Signal Action`” on page 486
- “`sigaddset—Add a Signal to a Signal Set`” on page 490
- “`sigemptyset—Initialize and Empty a Signal Set`” on page 492
- “`sigfillset—Initialize and Fill a Signal Set`” on page 493
- “`sigismember—See If a Signal Is a Member of a Signal Set`” on page 494
- “`signal—Install Signal Handler`” on page 495
- “`sigpending—Examine Pending Signals`” on page 498
- “`sigprocmask—Examine and Change Blocked Signals`” on page 499
- “`sigsuspend—Set Signal Mask and Wait for a Signal`” on page 502.

## sigemptyset—Initialize and Empty a Signal Set

This macro is used to initialize a signal set so that all supported signals are excluded. It is one of several macros that manage or query signal sets.

### Format

```
#include <signal.h>
int sigemptyset(sigset_t *set);
```

#### **set**

A pointer to the signal set to be initialized. The signal set is of the `sigset_t` type.

### Normal Return

The `sigemptyset` macro is always successful and returns a value of 0.

### Error Return

None.

### Programming Considerations

None.

### Examples

See “`sigaction`—Examine and Change Signal Action” on page 486 for an example of the `sigemptyset` macro.

### Related Information

- “`sigaction`—Examine and Change Signal Action” on page 486
- “`sigaddset`—Add a Signal to a Signal Set” on page 490
- “`sigdelset`—Delete a Signal from a Signal Set” on page 491
- “`sigfillset`—Initialize and Fill a Signal Set” on page 493
- “`sigismember`—See If a Signal Is a Member of a Signal Set” on page 494
- “`signal`—Install Signal Handler” on page 495
- “`sigpending`—Examine Pending Signals” on page 498
- “`sigprocmask`—Examine and Change Blocked Signals” on page 499
- “`sigsuspend`—Set Signal Mask and Wait for a Signal” on page 502.

---

## sigfillset—Initialize and Fill a Signal Set

This macro initializes a signal set so that it includes all supported signals. It is one of several macros that manage or query signal sets.

### Format

```
#include <signal.h>
int sigfillset(sigset_t *set);
```

#### set

A pointer to the signal set to be initialized. The signal set is of the `sigset_t` type.

### Normal Return

The `sigfillset` macro is always successful and returns a value of 0.

### Error Return

None.

### Programming Considerations

None.

### Examples

See “`sigprocmask`—Examine and Change Blocked Signals” on page 499 for an example of the `sigfillset` macro.

### Related Information

- “`sigaction`—Examine and Change Signal Action” on page 486
- “`sigaddset`—Add a Signal to a Signal Set” on page 490
- “`sigdelset`—Delete a Signal from a Signal Set” on page 491
- “`sigemptyset`—Initialize and Empty a Signal Set” on page 492
- “`sigismember`—See If a Signal Is a Member of a Signal Set” on page 494
- “`signal`—Install Signal Handler” on page 495
- “`sigpending`—Examine Pending Signals” on page 498
- “`sigprocmask`—Examine and Change Blocked Signals” on page 499
- “`sigsuspend`—Set Signal Mask and Wait for a Signal” on page 502.

---

## sigismember—See If a Signal Is a Member of a Signal Set

This macro determines whether a specific signal is a member of a signal set. It is one of several macros that manage or query signal sets.

### Format

```
#include <signal.h>
int sigismember(sigset_t *set,
                int signo);
```

#### **set**

A pointer to the signal set to be queried. The signal set is of the `sigset_t` type.

#### **signo**

One of the signals defined in the `signal.h` header file that corresponds to a specific signal to be queried. See Table 13 on page 495 for a list of supported signals.

### Normal Return

One of the following:

- If the signal specified by the **signo** parameter is a member of the signal set pointed to by the **set** parameter, the `sigismember` macro returns a value of 1.
- If the signal specified by the **signo** parameter is not a member of the signal set, the `sigismember` macro returns a value of 0.

### Error Return

If an error occurs, the `sigismember` macro returns a value of `-1` and sets `errno` to the following:

#### **EINVAL**

The value specified by the **signo** parameter is not a valid and supported signal number.

### Programming Considerations

None.

### Examples

See “sigpending—Examine Pending Signals” on page 498 for an example of the `sigismember` macro.

### Related Information

- “sigaction—Examine and Change Signal Action” on page 486
- “sigaddset—Add a Signal to a Signal Set” on page 490
- “sigdelset—Delete a Signal from a Signal Set” on page 491
- “sigemptyset—Initialize and Empty a Signal Set” on page 492
- “sigfillset—Initialize and Fill a Signal Set” on page 493
- “sigpending—Examine Pending Signals” on page 498
- “sigprocmask—Examine and Change Blocked Signals” on page 499
- “sigsuspend—Set Signal Mask and Wait for a Signal” on page 502.

## signal—Install Signal Handler

This function permits a process to choose one of several ways to handle a condition **sig** from the `raise`, `tpf_process_signals`, or `wait` function.

**Note:** The `signal` function is similar to the `sigaction` function.

### Format

```
#include <signal.h>
void(*signal (int sig, void(*func)(int)))(int);
```

#### sig

One of the signals defined in the `signal.h` header file. Table 13 lists the signals that are supported.

Table 13. Signals Supported in the C/C++ Environment

Value	Default Handler Action	Meaning
SIGABND	2	Abnormal end signal.
SIGABRT	1	Abnormal end (sent by the <code>abort</code> function).
SIGALRM	2	Timeout signal, such as is started by the <code>alarm</code> function.
SIGCHLD	3	Abnormal end, such as is initialized by the <code>abort</code> function.
SIGFPE	2	Arithmetic exceptions that are not masked; for example, overflow, division by zero, and incorrect operation.
SIGHUP	2	Hangup detected on controlling terminal or end of controlling process.
SIGILL	2	Detection of an incorrect function image.
SIGINT	2	Interactive attention.
SIGKILL	2	Termination signal. See note.
SIGPIPE	2	An attempt to write to a pipe when the pipe is not open for reading.
SIGSEGV	2	Incorrect access to storage.
SIGTERM	2	Termination request sent to the program. See note.
SIGUSR1	2	Intended for use by user applications.
SIGUSR2	2	Intended for use by user applications.
SIGIOERR	3	Intended for input/output (I/O) error.

#### Note:

The SIGKILL signal cannot be caught or ignored; therefore, it cannot be set using the `signal` function. If the `signal` function is coded to catch or ignore the SIGKILL signal, an error is returned. All signals, including the SIGKILL signal, remain pending (are not handled) by applications in the TPF system unless the `sleep`, `tpf_process_signals`, `wait`, or `waitpid` function is called.

In Table 13 the *default handler actions* are:

- 1 Abnormal termination of the entry control block (ECB).
- 2 Exit entry control block (ECB) with an OPR-7777 system error dump.
- 3 Ignored and control returns.

## signal

### **func**

One of the macros (SIG\_DFL or SIG\_IGN), defined in the `signal.h` header file, or a function address.

The action taken when the interrupt signal is received depends on the value of **func**.

Value	Meaning
<b>SIG_DFL</b>	Default handling for the signal occurs. See Table 13 on page 495 for default handler actions for each signal.
<b>SIG_IGN</b>	Ignore the signal. Control returns to the point where the signal was raised.

Otherwise, **func** points to a signal handler function.

When the signal is handled, the signal handler is reset to the default handler (the equivalent of `signal(sig, SIG_DFL);` is run) and **func** (the equivalent of `(*func)(sig);` is run. The **func** function can end by processing a return statement or by calling the `abort`, `exit`, or `longjmp` function. If **func** processes a return statement, the program resumes at the point the signal was raised.

## Normal Return

If successful, the call to `signal` returns the most recent value of **func**.

## Error Return

If there is an error in the call, the `signal` function returns a value of `-1` and a positive value in `errno`.

## Programming Considerations

- When either the `sigaction` or the `signal` function is used to change a signal action, the new signal action replaces the previous signal action regardless of whether the previous signal action was installed by a `sigaction` or `signal` function.
- The default setting is the handler, which is set for each of the signals during C environment initialization and when handler is reset on a call to the `raise` function.
- The default actions for SIG\_DFL are in segment CSIGDP. This can be modified to change SIG\_DFL actions for each of the signals system-wide.
- There is no static storage base switching support in activating the signal processor function set by calling the `signal` function. This does not cause problems when the signal processor is a library function or a DLM. Stub linkage handles static storage base switching.
- When the signal handler function is internal to the dynamic load module (DLM) where the `signal` function is called to set the handler, the `raise` function for that signal can only be called from the same DLM.
- The TPF system does not send any hardware signals. Signals can be sent by the `kill` function and the SIGCHLD signal can be sent to the parent of an exiting process that was created by the `tpf_fork` function.
- Signal handlers set to SIG\_IGN in the parent process are set to SIG\_IGN in the child process. Other handlers are set to SIG\_DFL in the child process.

## Examples

This example shows how to establish a signal handler.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#define ONE_K 1024
void StrCln(int);
void DoWork(char **, int);
void ADLM(int size) {
    char *buffer;
    if (signal((SIGUSR1), StrCln) == SIG_ERR) {
        printf("Could not install user signal");
        abort();
    }
    DoWork(&buffer, size);
    return;
}
void StrCln(int SIG_TYPE) {
    printf("Failed trying to malloc storage\n");
    return;
}
void DoWork(char **buffer, int size) {
    while (*buffer != NULL)
        *buffer = (char *)malloc(size*ONE_K);
    if (*buffer == NULL) {
        if (raise(SIGUSR1)) {
            printf("Could not raise user signal");
            abort();
        }
    }
    return;
}
```

## Related Information

- “abort—Terminate Program Abnormally” on page 7
- “raise—Raise Condition” on page 406
- “sigaction—Examine and Change Signal Action” on page 486
- “sigpending—Examine Pending Signals” on page 498
- “sigprocmask—Examine and Change Blocked Signals” on page 499
- “sigsuspend—Set Signal Mask and Wait for a Signal” on page 502.

---

## sigpending—Examine Pending Signals

This function allows the calling process to determine which signals have been sent to the calling process and are blocked from delivery.

### Format

```
#include <signal.h>
int sigpending(sigset_t *set);
```

#### set

A pointer to a signal set where the set of pending signals is stored by the sigpending function. The signal set is of the sigset\_t type.

### Normal Return

The sigpending function is always successful and returns a value of zero.

### Error Return

None.

### Programming Considerations

None.

### Examples

The following example queries pending signals.

```
#include <signal.h>
...
{
    sigset_t pending;

    /* initialize the new signal mask */
    sigpending(&pending);

    /* see if there's a SIGUSR1 signal pending */
    if (sigismember(&pending;, SIGUSR1))
    {
        /* SIGUSR1 is pending, do something */
    }
    ...
}
```

### Related Information

- “raise—Raise Condition” on page 406
- “sigaction—Examine and Change Signal Action” on page 486
- “sigismember—See If a Signal Is a Member of a Signal Set” on page 494
- “signal—Install Signal Handler” on page 495
- “sigprocmask—Examine and Change Blocked Signals” on page 499
- “sigsuspend—Set Signal Mask and Wait for a Signal” on page 502.



## sigprocmask—Examine and Change Blocked Signals

This function allows the calling process to examine (query) or change its signal mask. Each process has a signal mask that specifies a set of signals that cannot be raised. These are called blocked signals. A blocked signal can be sent to a process, but it remains pending until it is unblocked and subsequently raised.

### Format

```
#include <signal.h>
int sigprocmask( int          change,
                 const sigset_t *set,
                 sigset_t *oset);
```

#### change

If the signal set pointed to by the **set** parameter is not NULL, the **change** parameter indicates the way in which the signal mask (the set of signals currently blocked) is changed. This parameter must be specified as one of the following:

##### SIG\_BLOCK

All signals in the signal set pointed to by the **set** parameter are to be added to the current signal set for the process.

##### SIG\_SETMASK

The signal set pointed to by the **set** parameter replaces the current signal mask for the process.

##### SIG\_UNBLOCK

All signals in the signal set pointed to by the **set** parameter are to be removed from the current signal set for the process.

If the signal set pointed to by the **set** parameter is NULL, the **change** parameter is ignored.

#### set

One of the following:

- A pointer to a signal set that is used to modify the signal mask according to the **change** parameter. The signal set is of the `sigset_t` type.
  - If the **oset** parameter is a pointer to a signal set, the value of the signal mask before the `sigprocmask` function call is stored in the signal set referenced by the **oset** parameter, and the signal mask is changed as specified by the **change** and **set** parameters.
  - If the **oset** parameter is NULL, the signal mask is changed as specified by the **change** and **set** parameters.
- A value of NULL.
  - If the **oset** parameter is a pointer, the value of the signal mask stored in the signal set referenced by the **oset** parameter and the signal mask is not changed.
  - If the **oset** parameter is NULL, the signal mask is not examined or changed.

#### oset

One of the following:

- A pointer to a signal set where the signal mask for the process is stored when the `sigprocmask` function is called. The signal set is of the `sigset_t` type.
- A value of NULL.

## sigprocmask

**Note:** The description of the **set** parameter describes the results of sigprocmask function processing based on the value specified for the **oset** parameter.

## Normal Return

If successful, the sigprocmask function returns a value of zero.

## Error Return

If unsuccessful, the signal mask for the process is not changed and the sigprocmask function returns a value of -1 and sets errno to the following:

### EINVAL

The value of the **change** parameter is not valid; it must be one of the values listed for the **change** parameter.

## Programming Considerations

- The signal mask is managed by the sigprocmask function and also by the TPF system.
- The sigprocmask function is intended only for use in a single-threaded process.
- To query the current signal mask without changing it, specify:
  - NULL for the **set** parameter
  - A pointer to a signal set for the **oset** parameter.

The results of the query are in the signal set referenced by the **oset** parameter.

## Examples

The following example shows how to block all signals.

```
#include <signal.h>
...

{
    struct sigaction      act, oact;

    sigset_t new_mask;
    sigset_t old_mask;

    /* initialize the new signal mask */
    sigfillset(&new_mask);

    /* block all signals */
    sigprocmask(SIG_SETMASK, &new_mask, &old_mask);

    /* call function */
    somefunc();

    /* restore signal mask */
    sigprocmask(SIG_SETMASK, &old_mask, NULL);
    ...
}
```

## Related Information

- “raise—Raise Condition” on page 406
- “sigaction—Examine and Change Signal Action” on page 486
- “sigfillset—Initialize and Fill a Signal Set” on page 493
- “signal—Install Signal Handler” on page 495

- “sigpending—Examine Pending Signals” on page 498
- “sigsuspend—Set Signal Mask and Wait for a Signal” on page 502.

## sigsuspend–Set Signal Mask and Wait for a Signal

This function replaces the current signal mask for the calling process with a new signal set and then suspends the calling process until a signal is delivered.

### Format

```
#include <signal.h>
int sigsuspend(sigset_t *set);
```

#### set

A pointer to the signal set that replaces the current signal mask. The signal set is of the sigset\_t data type.

### Normal Return

The purpose of the sigsuspend function is to suspend the calling process until a signal causes control to be returned. When a signal is received while the calling process is suspended and the associated signal handler returns control to the calling process, the sigsuspend function returns a value of `-1` and sets `errno` to **EINTR**, which indicates that sigsuspend function processing was interrupted by a signal.

### Error Return

None.

## Programming Considerations

- sigsuspend function processing changes the signal mask of the calling process according to the sigset\_t data type referenced by the **set** parameter. However, on return to the calling process, the signal mask is restored to the value it had before the call to the sigsuspend function.
- When a signal that interrupts sigsuspend function processing is received, the signal is handled. If the entry control block (ECB) exits while handling the signal, control is not returned to the calling process.

## Examples

The following example suspends processing of the current process until a signal arrives.

```
#include <signal.h>
:

{
    sigset_t new_mask;

    /* initialize the new signal mask */
    sigemptyset(&new_mask);
    sigaddset(&new_mask, SIGCHLD);

    /* wait for any signal except SIGCHLD */
    sigsuspend(&new_mask);

    /* we only expect to get control here if sigsuspend was */
    /* interrupted by a signal for which a signal handler */
    /* was called and only if that signal handler returned */
    :

    /* initialize the new signal mask */
    sigfillset(&new_mask);
    sigdelset(&new_mask, SIGUSR1);
```

```
/* wait for ONLY a SIGUSR1 signal */
sigsuspend(&new_mask);

/* we only expect to get control here if sigsuspend was */
/* interrupted by a signal for which a signal handler */
/* was called and only if that signal handler returned */
...
}
```

## Related Information

- “pause—Wait for a Signal” on page 388
- “raise—Raise Condition” on page 406
- “sigaction—Examine and Change Signal Action” on page 486
- “sigaddset—Add a Signal to a Signal Set” on page 490
- “sigdelset—Delete a Signal from a Signal Set” on page 491
- “sigemptyset—Initialize and Empty a Signal Set” on page 492
- “sigfillset—Initialize and Fill a Signal Set” on page 493
- “signal—Install Signal Handler” on page 495
- “sigpending—Examine Pending Signals” on page 498
- “sigprocmask—Examine and Change Blocked Signals” on page 499.

## sipcc–System Interprocessor Communication

This function provides communication among processors in a TPF loosely coupled (LC) environment. It causes an interprocessor communication (IPC) item to be transmitted to a specified program segment in one or all active I-streams, and in one or more active processors. The destination processors can be specified as a single processor, as a list of processors, or as a broadcast to all processors. The destination I-streams can be targeted as either the main I-stream or all I-streams in the destination processors.

An IPC item consists of control information and, optionally, two variable length data areas. Data area 1 (DA1) may be up to 104 bytes maximum. Data area 2 (DA2) may be any length that will fit in a single 128-, 381-, 1055-, or 4K-byte storage block.

### Format

```
#include <sysapi.h>
int sipcc(const struct sipcc_parm *sipcc_parm_list);
```

#### **sipcc\_parm\_list**

A pointer to a sipcc\_parm structure as described below:

#### **sipcc\_num\_parm**

Number of parameters following this one. Values between SIPCC\_PARM\_MAX and SIPCC\_PARM\_MIN are valid. This number defines the range of parameters that are recognized by the sipcc handling routine. The range always begins with the first parameter (sipcc\_receiving\_pgm). The following parameters are in the exact order that will be described by the range.

#### **sipcc\_receiving\_pgm**

Program name that is invoked in the receiving processors. It is a required parameter.

#### **sipcc\_da1\_len**

Length of data area 1. Maximum length is 104 bytes. It is an optional parameter.

#### **sipcc\_da1\_ptr**

Pointer to data area 1. This parameter is ignored when sipcc\_da1\_length is zero. It is required when sipcc\_da1\_length is not zero.

#### **sipcc\_da2\_len**

Length of data area 2. The length cannot exceed the size of the storage block specified in the level of sipcc\_da2\_. It is an optional parameter.

#### **sipcc\_da2\_lvl**

A valid data level from enumeration type t\_lvl. This field is ignored if sipcc\_da2\_length is zero. It is required when sipcc\_da2\_length is not zero. If a data level is specified, that data level must hold a block.

#### **sipcc\_priority**

SIPCC\_PRIORITY\_YES for a high priority request or SIPCC\_PRIORITY\_NO for a regular request. SIPCC\_PRIORITY\_NO is the default if the parameter is omitted.

#### **sipcc\_xmit\_type**

SIPCC\_XMIT\_IP for inter-processor or SIPCC\_XMIT\_IS for inter-processor/inter-I-stream transmission. SIPCC\_XMIT\_IP is the default if the parameter is omitted.

**sipcc\_receiving\_proc**

The processors that are to receive this item. For a single processor, specify the ordinal number of the processor; for all active processors, specify `SIPCC_ALL_ACTIVE_PROC`; for a list of processors, specify `SIPCC_PROCESSOR_LIST`. If a list is specified, **sipcc\_list\_ptr** must contain a pointer to the list of processors. The default is `SIPCC_ALL_ACTIVE_PROC` if the parameter is omitted.

**sipcc\_receiving\_is**

I-streams to be receiving the item. `SIPCC_IS_MAIN` or `SIPCC_IS_ALL` are valid. `SIPCC_IS_MAIN` is the default if the parameter is omitted.

**sipcc\_response**

`SIPCC_RESPONSE_YES` or `SIPCC_RESPONSE_NO`. This parameter is only used by user application programs. The SIPCC service routine will only pass the value to user applications without any processing. The default value is `SIPCC_RESPONSE_NO` if the parameter is omitted.

**sipcc\_item\_type**

`SIPCC_REQUEST_ITEM` or `SIPCC_RESPONSE_ITEM`. This parameter is only used by user application programs. The SIPCC service routine will only pass the value to user applications without any processing. The default value is `SIPCC_REQUEST_ITEM` if the parameter is omitted.

**sipcc\_list\_ptr**

A pointer to an area of storage containing a list of destination processor ordinal numbers. The **sipcc\_list\_ptr** parameter is required when specifying a list of destination processors (the **sipcc\_receiving\_proc** parameter is set to `SIPCC_PROCESSOR_LIST`) and is optional for requests that broadcast to all processors or that transmit to a single processor. If specified, on return from the request the list area contains a list of processor ordinal numbers for all processors to which transmission was started.

## Normal Return

`SIPCC_OK` is returned when transmission to the destination processor has been initiated.

## Error Return

`SIPCC_FAILED` is returned when transmission to the destination processor was not initiated.

## Programming Considerations

- This function can be issued only by an E-type program.
- The C library function service routine will set up the control area, which includes the destination, flags, and length fields of data area 1 and data area 2, and issues a SIPCC SVC.
- The address of the optional second data area, DA2, is specified in the storage block reference word of the ECB at the level specified in the parameter list. DA2 must reside in a storage block. If DA2 exists, the data transmitted will begin at byte 0 of the storage block for the specified length.
- The storage block provided for data area 2 will remain attached to the ECB upon return of control at NSI.
- There will be no attempt to transmit to an inactive processor.
- IPC items that require expedited handling are identified by setting the **SIPCC\_PRIORITY\_YES** flag in the parameter list. Priority requests cause incore

## sipcc

staging to be halted and transmission for all available items to be initiated. In the receiving processors, priority items are placed on the ready list.

- The program segment named in the parameter list (the program segment that is invoked in the receiving processor) must reside on the database indicated by the PBI field in the requestor's ECB.
- The PBI, DBI, and SSU ID fields in the requester's ECB will be transmitted to the destination processors. There, they will be used to initialize the ECB and invoke the specified program.
- The format of an IPC item as seen by the destination program segment in the receiving processor is as follows:
  - Data area 1 will be placed in the first work area of the ECB beginning at location EBW000.
  - Data area 2 will be placed in a storage block on level 0 of the ECB. The size of the storage block will be the same as the sender's block size.
- There is an implied Wait if storage blocks required by the System Interprocessor Communication Facility (SICF) are not available when needed.
- The sipcc library function routine will recognize the following sipcc error and exit the ECB
  - The length of DA1 is greater than 104 bytes.
  - The length of DA2 is greater than the storage block size on the specified level.
  - The **sipcc\_list\_ptr** parameter must contain a pointer to a list of processors, but the value is null.
- If **sipcc\_list\_ptr** is specified, the original content of the list area is not preserved. The list area is overlaid by the sipcc function with a list of processor ordinal numbers to which transmission was started.

## Examples

The following example calls sipcc to broadcast an IPC to program ABCD in all active processors at regular priority, with no response required. 200 bytes of data reside in a data block on data level D4.

```

:
struct sipcc_parm  sipcc_parm_list = {
    5,                /* 5 parameters follow.      */
    { "ABCD" }, /* #1 - Receiving pgm name.  */
    0,                /* #2 - Data area 1 length.  */
    NULL,             /* #3 - Data area 1 address. */
    200,              /* #4 - Data area 2 length.  */
    D4                /* #5 - Data area 2 data lvl.*/
};
:
if (sipcc(&sipcc_parm_list) != SIPCC_OK)
{
    /* Handle sipcc failure. */
}
```

## Related Information

See *TPF System Macros* for information about the SIPCC macro.



## sleep—Suspend the Calling Process

This function causes the calling process to be suspended until the specified amount of real time has elapsed or a signal is caught by the process and the signal handler returns.

### Format

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

#### seconds

The number of seconds of sleep time. The maximum number of seconds is 65 535. A higher value can be specified, but a value of 65 535 will actually be used.

### Normal Return

The `sleep` function is always successful.

If the `sleep` function returns because of the receipt of a signal, the return value is the number of seconds remaining. Otherwise, 0 is returned.

### Error Return

Not applicable.

## Programming Considerations

- If a signal is received while the program is sleeping, that signal is raised; that is, if the `signal` function was used to install signal handlers, the placement of the `sleep` function in the code is subject to the same restrictions as the `raise` function.
- `sleep` time is not exact; for example, the process would have to wait until its I-stream becomes available after the `sleep` timer expires.

## Examples

The following example waits for an incoming signal or waits up to 10 seconds.

```
#include <unistd.h>
...
/* Wait for an incoming signal or up to 10 seconds */

sleep (10);
...
```

## Related Information

- “alarm—Schedule an Alarm” on page 13
- “kill—Send a Signal to a Process” on page 298
- “raise—Raise Condition” on page 406
- “signal—Install Signal Handler” on page 495
- “tpf\_fork—Create a Child Process” on page 594.

## snapc—Issue Snapshot Dump

This function causes the system error routine to issue a snapshot dump and display, optionally, a message (pointed to by **msg**) at the prime CRAS. Code **action**, as either **SNAPC\_EXIT** to exit the ECB or as **SNAPC\_RETURN** to return to the calling program.

### Format

```
#include <tpfapi.h>
void snapc(int action, int code, const char *msg,
           struct snapc_list **listc, char prefix, int regs,
           int ecb, const char *program);
```

#### action

The action of the ECB after the snapshot dump. This argument is an integer. Code the defined term **SNAPC\_EXIT** to force the ECB to exit, or **SNAPC\_RETURN** to cause a return to the calling program.

#### code

The identification number for the snapshot dump. This argument is an integer and should be a unique number ranging from 0 to X'7FFFFFFF'.

#### msg

This argument is a pointer to type char, which is a message text string to be displayed at the CRAS console and appended to the dump. The string must be terminated by \0 and must not exceed 255 characters.

If no message is desired, code the defined term **NULL**.

#### listc

This argument is a pointer to an array of pointers that point to type struct snapc\_list, indicating areas of storage to be displayed on the dump. The snapc\_list structure has four fields:

##### snapc\_len

type short int, the length of the area to be dumped. Code zero for snapc\_len to indicate no more areas are to be dumped.

##### snapc\_name

character string, the name of the data area. snapc\_name must be 8 characters long, left justified, and padded with blanks.

##### snapc\_tag

the location of the area to be dumped.

##### snapc\_indir

whether the address in snapc\_tag is an indirection. To indicate an indirection, code the defined term **SNAPC\_INDIR**. If snapc\_tag is not an indirection, code the defined term **SNAPC\_NOINDIR**.

If no storage list exists, code the defined term **NULL**.

**Note:** There is a limit of 50 entries in the snapc\_list array.

#### prefix

Specifies the prefix of the snapshot dump code. This argument is a character. It allows each application to have its own SNAPC code name space. You can use any uppercase alphabetic characters in the following ranges: A-H and J-V. The letters I and W—Z are reserved for IBM use. There is no default for this parameter.

**regs**

Code the defined term **SNAPC\_REGS** to include the registers in the snapshot dump, or **SNAPC\_NOREGS** to exclude the registers. This argument is an integer.

**Note:** **SNAPC\_REGS** is ignored for ISO-C.

**ecb**

Code the defined term **SNAPC\_NOECB** to force the names of the subsystem and subsystem user to that of the basic subsystem. This renders the terminal address not available (N.A.). Code the defined term **SNAPC\_ECB** to take the subsystem and subsystem user names and the terminal address from the ECB. This argument is an integer.

**Note:** **SNAPC\_NOECB** is ignored for ISO-C.

**program**

This argument is a pointer to type char, which is a text string indicating the program name to be used in the snapshot dump. The string must end in a -0 and should not be more than 16 characters. If the string exceeds this length it will be truncated. If coded as NULL, the program name is taken from the current program.

**Normal Return**

Void.

**Error Return**

Not applicable.

**Programming Considerations**

None.

**Examples**

The following example forces a snapshot dump bearing ID number 12345 and a prefix of A to be issued. The registers will be included in the dump and control will be returned to the program after the dump. The ECB will be used for the subsystem and subsystem user names, and terminal ID and the program name will be C001. The snapc\_list is snapstuff and the message will be PROGRAM BLEW UP.

```
#include <tpfeq.h>
#include <tpfapi.h>

test()
{
    struct snapc_list *snapstuff[3],list[3];

    snapstuff[0]=(struct snapc_list *) &list[0];
    snapstuff[1]=(struct snapc_list *) &list[1];
    snapstuff[2]=(struct snapc_list *) &list[2];

    /* Dump the 4 bytes of data in EBW000-EBW003. */
    snapstuff[0]->snapc_len = 4;
    snapstuff[0]->snapc_name = "MYSTUFF ";
    snapstuff[0]->snapc_tag = &ecbptr()->ebw000;
    snapstuff[0]->snapc_indir = SNAPC_NOINDIR;

    /* Dump the first 100 bytes of the core block on data level D0. */
    snapstuff[1]->snapc_len = 100;
    snapstuff[1]->snapc_name = "DATA100 ";
```

## snapc

```
snapstuff[1]->snapc_tag = &ecbptr()->celcr0;  
snapstuff[1]->snapc_indir = SNAPC_INDIR;  
  
snapstuff[2]->snapc_len = 0;  
  
snapc(SNAPC_RETURN, 0x12345, "PROGRAM BLEW UP", snapstuff,  
      'A', SNAPC_REGS, SNAPC_ECB, "C001");  
  
exit(0);  
}
```

## Related Information

- “serrc\_op—Issue System Error: Operational” on page 454
- *TPF General Macros*, SNAPC macro.

## sonic—Obtain Symbolic File Address Information

Given a symbolic file address this function returns the file address reference format (FARF<sub>x</sub>) addressing mode and record type of that file address.

### Format

```
#include <tpfapi.h>
unsigned long sonic(enum t_lvl level );
```

or

```
#include <tpfapi.h>
unsigned long sonic(TPF_FA8 *fa8 );
```

#### level

A file address reference word (FARW) (D0–DF) containing the symbolic file address being queried.

#### fa8

A pointer to an 8-byte file address.

### Normal Return

Non-zero, with bit settings as described in Table 14.

Table 14. sonic Normal Return

Bit Masks	Bit	Description
#define SONIC_TYPE_INDICATOR 0x80000000	00	0 = fixed Nonzero = pool
#define SONIC_POOL_LONGEVITY 0x40000000	01	0 = long-term pool 1 = short-term pool (See note 1 on page 512.)
#define SONIC_ADDRESS_FORMAT 0x30000000	02-03	00 = FARF3 addressing format 01 = FARF4 addressing format 11 = FARF5 addressing format 10 = FARF6 addressing format
	04-06	0
#define SONIC_FADDR_INVALID 0x01000000	07	0 = Valid file address 1 = Invalid file address
	08-15	0
#define SONIC_RECORD_UNIQUE 0x00008000	16	0 = address is common to all SSUs/ processors/l-streams of the caller's subsystem 1 = address is unique to some combination of SSUs/processors/ l-streams in the subsystem. (See note 2 on page 512.)
#define SONIC_ALT_REC_SIZE 0x00004000	17	0 = size is indicated by bit 31 alone 1 = size is 4K (4096) bytes Bit 31 should also = 1
	18-27	0
Pool address indicator	28	0 = fixed address 1 = pool address (See note 3 on page 512.)
#define SONIC_DUPLICATED 0x00000004	29	0 = not duplicated/no fallback 1 = duplicated/fallback

## sonic

Table 14. sonic Normal Return (continued)

Bit Masks	Bit	Description
	30	Always set to 1
#define SONIC_REC_SIZE 0x00000001	31	0 = small size record (381 bytes) 1 = large size record (1055 or 4096 bytes)
<b>Notes:</b> 1. This bit is meaningful only when the file address is a pool address (that is, bit 0 = 1); otherwise, it is zero. 2. This bit is meaningful only when the file address is a fixed address (that is, bit 0 = 0). 3. This bit has no equate associated with it. It is always set to 1 to maintain compatibility with an earlier interface.		

## Error Return

The **SONIC\_ERROR** tag is defined for interrogating the error return.

### SONIC\_ERROR

The return code if the FARF address input is invalid.

SNAPC with the error number OPR-074 will be issued when NO\_LVL was specified for the one required parameter, or if an invalid data level value was encountered. The ECB is exited when the dump is taken.

## Programming Considerations

- The symbolic file address must be contained in the file address reference word (FARW), CE1FM(x), where (x) is the specified level.
- The FARW at the specified level is unchanged.
- Applications that call this function using 8-byte file addresses instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example returns the file characteristics of the FARF address in data level 4.

```
#include <tpfapi.h>
unsigned long ret_code;
:
ret_code = sonic(D4);
```

The following example returns the file characteristics of the 8-byte FARF address.

```
#include <tpfapi.h>
unsigned long ret_code;
TPF_FA8 fa8;
:
ret_code = sonic(&fa8);
```

## Related Information

“tpf\_esfac—Obtain Extended Symbolic File Address Information” on page 586

## stat—Get File Information

This function gets file information.

### Format

```
#include <sys/stat.h>
int stat(const char *pathname, struct stat *info);
```

#### pathname

A pointer to the path name of the file or directory.

#### info

A pointer to the struct stat object in which file status information will be returned.

This function gets status information about a specified file and places it in the area of storage pointed to by the **info** argument. The process does not need permissions on the file itself, but must have search permission on all directory components of **pathname**. If the named file is a symbolic link, the **stat** function resolves the symbolic link. It also returns information about the resulting file.

The information is returned as shown in the following stat structure table as defined in the `sys/stat.h` header file.

Table 15. Elements of the stat Structure

Structure	Description
mode_t st_mode	A bit string indicating the permissions and privileges of the file. Symbols are defined in the <code>sys/stat.h</code> header file to refer to bits in a <code>mode_t</code> value; these symbols are listed in “ <a href="#">chmod—Change the Mode of a File or Directory</a> ” on page 32.
ino_t st_ino	The serial number of the file.
dev_t st_dev	The numeric ID of the device containing the file. In the TPF file system, this is the subsystem ID.
nlink_t st_nlink	The number of links to the file.
uid_t st_uid	The numeric user ID of the file owner.
gid_t st_gid	The numeric group ID of the file group.
off_t st_size	For regular files, this field contains the file size in bytes. For other kinds of files where the file size cannot be determined, the value of this field is set to <code>-1</code> . For directories, this field contains the number of entries in the directory.
time_t st_atime	The TPF system does not update <code>st_atime</code> . It contains <code>(time_t) - 1</code> unless the file's access time stamp has been updated by the <code>utime</code> function.
time_t st_ctime	The TPF system does not update <code>st_ctime</code> . It always contains the value <code>(time_t) - 1</code> .
time_t st_mtime	The most recent time the contents of the file were changed given in terms of seconds that have elapsed since epoch.

#### TPF deviation from POSIX

The TPF system updates the `mtime` time stamp only when a file that was previously opened for write access, or for read and write access, is closed.

Table 15. Elements of the stat Structure (continued)

Structure	Description
st_blksize	The preferred block size. In the TPF file system, this field is always set to 1. This does not imply anything about the underlying structure of the TPF file system. This field is provided for Berkeley Software Distribution (BSD) compatibility only.
st_blocks	The number of blocks used by the file. In the TPF file system, this value is equal to the size of the file. This field is provided for BSD compatibility only.
st_atimespec	The time the file was last accessed, expressed as a struct timespec (seconds since epoch plus nanoseconds). The value of the st_atimespec field has the same restrictions as the st_atime field. This field is provided for BSD compatibility only.
st_ctimespec	The time the file status last changed, expressed as struct timespec (seconds since epoch plus nanoseconds). The value of the st_ctimespec field has the same restrictions as the st_ctime field. This field is provided for BSD compatibility only.
st_mtimespec	The time the file data last changed, expressed as struct timespec (seconds since epoch plus nanoseconds). The value of the st_mtimespec field has the same restrictions as the st_mtime field. This field is provided for BSD compatibility only.

**TPF deviation from POSIX**

TPF does not support the atime (access time) or ctime (change time) time stamp.

Values for time\_t are given in terms of seconds since epoch.

You can examine properties of a mode\_t value from the st\_mode field using a collection of macros defined in the modes.h header file. If **mode** is a mode\_t value:

**S\_ISBLK(mode)** Is nonzero for block special files.

**TPF deviation from POSIX**

The TPF system does not support block special files; this macro is included for compatibility only.

**S\_ISCHR(mode)** Is nonzero for character special files.

**S\_ISDIR(mode)** Is nonzero for directories.

**S\_ISFIFO(mode)** Is nonzero for pipes and first-in-first-out (FIFO) special files.

**S\_ISLNK(mode)** Is nonzero for symbolic links.

**S\_ISREG(mode)** Is nonzero for regular files.

**S\_ISSOCK(mode)** Is nonzero for sockets.

If the stat function successfully determines this information, it stores the information in the area indicated by the **info** argument.



## Normal Return

If successful, the stat function returns a value of zero.

## Error Return

If unsuccessful, the stat function returns a value of `-1` and sets `errno` to one of the following:

<b>EACCES</b>	The process does not have search permission on some component of the <b>pathname</b> prefix.
<b>EINVAL</b>	<b>info</b> is a null pointer.
<b>ELOOP</b>	This error is issued if the number of symbolic links found while resolving the <b>pathname</b> argument is greater than <code>POSIX_SYMLINK_MAX</code> .
<b>ENAMETOOLONG</b>	<b>pathname</b> is longer than <code>PATH_MAX</code> characters or some component of <b>pathname</b> is longer than <code>NAME_MAX</code> characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds <code>PATH_MAX</code> .
<b>ENOENT</b>	There is no file named <b>pathname</b> , or <b>pathname</b> is an empty string.
<b>ENOTDIR</b>	A component of the <b>pathname</b> prefix is not a directory.

## Programming Considerations

None.

## Examples

The following example gets status information about a file.

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

main() {
    struct stat info;

    if (stat("/", &info) != 0)
        perror("stat() error");
    else {
        puts("stat() returned the following information about root f/s:");
        printf("  inode:  %d\n", (int) info.st_ino);
        printf("  mode:   %08x\n", info.st_mode);
        printf("  links:  %d\n", info.st_nlink);
        printf("  uid:    %d\n", (int) info.st_uid);
        printf("  gid:    %d\n", (int) info.st_gid);
        printf("modified: %s", ctime(&info.st_mtime));
    }
}
```

### Output

```
stat() returned the following information about root f/s:
inode:    0
mode:     010001ed
links:    0
uid:      0
gid:      1
modified: Fri Jan 16 10:07:55 1998
```

**stat**

## **Related Information**

- “remove—Delete a File” on page 427
- “chmod—Change the Mode of a File or Directory” on page 32
- “chown—Change the Owner or Group of a File or Directory” on page 35
- “creat—Create a New File or Rewrite an Existing File” on page 54
- “fstat—Get Status Information about a File” on page 230
- “link—Create a Link to a File” on page 302
- “lstat—Get Status of a File or Symbolic Link” on page 317
- “mkdir—Make a Directory” on page 326
- “open—Open a File” on page 380
- “read—Read from a File” on page 410
- “symlink—Create a Symbolic Link to a Path Name” on page 521
- “unlink—Remove a Directory Entry” on page 668
- “write—Write Data to a File Descriptor” on page 696.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## strerror–Get Pointer to Run-Time Error Message

This function maps the error number in the **errno** field to an error message string. The error number must be a valid errno value.

### Format

```
#include <string.h>
char *strerror(int errno);
```

#### **errno**

A valid errno value.

### Normal Return

This function returns a pointer to the string, which may be overwritten by a subsequent call to the strerror function. Do not let the program modify the contents of this string.

### Error Return

Not applicable.

### Programming Considerations

The strings that are returned by the strerror function are defined in segment CBSTER. User modifications to these strings will affect strerror output.

### Examples

The following example calls the sqrt() function and prints an error message if an error occurs.

```
#include <math.h>
#include <string.h>
#include <errno.h>

void test(double d)
{
    double r = 0.0;
    errno = 0;
    r = sqrt(d);
    if (errno != 0)
        printf("Error in sqrt: %s\n", strerror(errno));
}
```

### Related Information

None.

## swisc\_create—Create New ECB on Specified I-Stream

This function creates a new ECB on another I-stream.

### Format

```
#include <sysapi.h>
void swisc_create(struct swisc_create_input *create_parameters);
```

or

```
#include <sysapi.h>
void swisc_create(struct swisc_create_input *create_parameters,
                 TPF_DECB *decb);
```

#### create\_parameters

A pointer to a fully initialized `swisc_create_input` structure. The following fields are defined in struct `swisc_create_input`:

##### program

A pointer to the name of the first program segment to be executed by the created ECB. This can be a DLM, a BAL segment, or a TARGET(TPF) C segment. Data can only be passed to **program** via the EBW work area or via a core block (see **ebw\_data\_length**, **ebw\_data**, and **data\_level** below).

##### istream

The I-stream where the new entry is to be created; either the I-stream (1-based ordinal) number, or one of the following:

##### SWISC\_IS\_MAIN

Create the new entry on the main I-stream.

##### SWISC\_IS\_MPIF

Create the new entry on the MPIF I-stream.

##### SWISC\_IS\_BALANCE

Create the new entry on the least busy I-stream.

##### cpu\_list

One of:

##### SWISC\_LIST\_DEFER

Add the new entry to the deferred list.

##### SWISC\_LIST\_INPUT

Add the new entry to the input list.

##### SWISC\_LIST\_READY

Add the new entry to the ready list.

##### bypass

One of:

##### SWISC\_BYPASS\_NO

The `istream` value is limited to the I-streams usable by applications, as controlled by the main supervisor command ZCNIS.

##### SWISC\_BYPASS\_YES

The `istream` value is limited to all the I-streams online (active) in the CPC.

##### ebw\_data\_length

The number of bytes of data to be passed to the new entry, from 0 to 104.

**ebw\_data**

A pointer to void containing the address of data to be passed to the new entry beginning at EBW000.

**data\_level**

One of 16 possible values representing a valid data level from enumeration type `t_lvl`, expressed as **Dx**, where *x* represents the hexadecimal number of the level (0–F). The data on data level **Dx** is passed to the new entry. If no data level data is to be passed to the new entry, code **NO\_LVL**.

**decb**

A pointer of type `TPF_DECB *`, representing a valid data event control block (DECB). The working storage block on the specified DECB is passed to the new entry. If a DECB is specified, the **data\_level** field in **create\_parameters** must be set to the value **NO\_LVL**.

## Normal Return

Void.

## Error Return

Not applicable.

## Programming Considerations

- This macro can be executed on any I-stream.
- If **data\_level** is not set to **NO\_LVL**, the data level specified must have an attached block. This block will be unhooked from the issuing ECB upon return from the `swisc_create` function.
- If **decb** is specified, the DECB must have an attached block. This block will be unhooked from the issuing ECB on return from the `swisc_create` function.
- Only a single working storage block can be passed to the new entry; therefore, if **decb** is not set to NULL and **data\_level** is not set to **NO\_LVL**, control is transferred to the system error routine and the issuing ECB will be exited.
- No I/O should be outstanding when calling this function.
- System functions required to act on physical I-streams must set bypass to **SWISC\_BYPASS\_YES**.
- If `istream` is set to **SWISC\_IS\_BALANCE**, the TPF scheduler will be invoked.
- ECBs may or may not be started on the same I-stream where they are created, depending on the options set by the programmer and whether the scheduler is invoked.
- In addition to the normal macro trace information, the trace entry resulting from executing `swisc_create` will contain the activation type as CREATE, the name of the target program and the target I-stream number.
- If you use this function to create an ECB that will enter a dynamic load module (DLM) with an entry point defined by the `main` function, the TPF system assumes that any core block attached to data level 0 (D0) contains a command string that will be parsed into `argc` and `argv` parameters for the `main` function. See *TPF Application Programming* for more information about the `main` function.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## swisc\_create

### Examples

The following example creates a new entry and adds it to the ready list for whichever application I-stream is least active. No data is passed to the new entry. When the new entry is gets control of the selected I-stream, it will execute program XYZ1.

```
#include <sysapi.h>
:
:
struct swisc_create_input create_parameters;
extern void XYZ1(void);

create_parameters.program = "XYZ1";
create_parameters.istream = SWISC_IS_BALANCE;
create_parameters.cpu_list = SWISC_LIST_READY;
create_parameters.bypass = SWISC_BYPASS_NO;
create_parameters.ebw_data = NULL;
create_parameters.ebw_data_length = 0;
create_parameters.data_level = NO_LVL;

swisc_create(&create_parameters);
```

The following example creates a new entry and adds it to the deferred list for the main I-stream. No data is passed to the new entry; however, the core block attached at the specified DECB is unhooked from the issuing ECB and passed to the new entry. When the new entry gets control of the selected I-stream, it will run program XYZ1.

```
#include <sysapi.h>
:
:
struct swisc_create_input create_parameters;
extern void XYZ1(void);
TPF_DECB *decb;

create_parameters.program = "XYZ1";
create_parameters.istream = SWISC_IS_MAIN;
create_parameters.cpu_list = SWISC_LIST_DEFER;
create_parameters.bypass = SWISC_BYPASS_NO;
create_parameters.ebw_data = NULL;
create_parameters.ebw_data_length = 0;
create_parameters.data_level = NO_LVL;

swisc_create(&create_parameters, decb);
```

### Related Information

“creec, \_\_CREEC—Create a New ECB with an Attached Block” on page 59

For more information about DECBs, see *TPF Application Programming*.

## symlink—Create a Symbolic Link to a Path Name

This function creates a symbolic link to a path name.

### Format

```
#include <unistd.h>
int symlink(const char *pathname, const char *slink);
```

#### **pathname**

A pointer to the path name to which the symbolic link resolves.

#### **slink**

The name of the symbolic link.

This function creates the symbolic link named by **slink** with the file specified by **pathname**. File access checking is not performed on the **pathname** file and the file does not need to exist.

A symbolic link path name is resolved as follows:

- When a component of a path name refers to a symbolic link rather than to a directory, the path name contained in the symbolic link is resolved.
- If the path name in the symbolic link begins with a slash (/), the symbolic link path name is resolved with respect to the root directory.  
If the path name in the symbolic link does not start with a slash (/), the symbolic link path name is resolved with respect to the directory that contains the symbolic link.
- If the symbolic link is the last component of a path name, it may or may not be resolved. Resolving the path name depends on the function using the path name. For example, the rename function does not resolve a symbolic link when it appears as the final component of either the new or old path name. However, the open function does resolve a symbolic link when it appears as the last component.
- If the symbolic link is not the last component of the original path name, remaining components of the original path name are resolved with respect to the symbolic link.

Because the mode of a symbolic link cannot be changed, its mode is ignored during the lookup process. Any files and directories to which a symbolic link refers are checked for access permission.

### Normal Return

If successful, the `symlink` function returns a value of zero.

### Error Return

If unsuccessful, the `symlink` function returns a value of `-1`, does not affect any file it names, and sets `errno` to one of the following:

**EACCES**      A component of the **slink** path prefix denies search permission, or write permission is denied in the parent directory of the symbolic link to be created.

**EINVAL**      This may be returned for the following reason:

- **pathname** is an empty string.

**ENAMETOOLONG**

**pathname** is longer than `PATH_MAX` characters or some

## symlink

component of **pathname** is longer than NAME\_MAX characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH\_MAX.

<b>ENOTDIR</b>	A component of the path prefix of <b>slink</b> is not a directory.
<b>ELOOP</b>	A loop exists in symbolic links. This error is issued if the number of symbolic links found while resolving the <b>slink</b> argument is greater than POSIX_SYMLINK_MAX.
<b>EEXIST</b>	The file named by <b>slink</b> already exists.
<b>ENOSPC</b>	The new symbolic link cannot be created because there is no space left on the file system that will contain the symbolic link.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

Because the symlink function requires an update to a directory file, new symbolic links cannot be created in 1052 or UTIL state.

## Examples

The following example provides symlink information for a file.

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

main() {
    char fn[]="test.file";
    char sln[]="test.symlink";
    int fd;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        if (symlink(fn, sln) != 0) {
            perror("symlink() error");
            unlink(fn);
        }
        else {
            unlink(fn);
            unlink(sln);
        }
    }
}
```

## Related Information

- “link—Create a Link to a File” on page 302
- “readlink—Read the Value of a Symbolic Link” on page 417
- “unlink—Remove a Directory Entry” on page 668.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



## syslog—Send a Message to the Control Log

This function sends a message to the syslog daemon, which writes the message to an appropriate system log, forwards it to a list of users, or forwards it to the logging facility on another host over the network.

See *TPF Transmission Control Protocol/Internet Protocol* for more information about the syslog daemon.

### Format

```
#include <syslog.h>
void    syslog(int priority, const char *message, ... /* argument */);
```

#### priority

Specify this argument by ORing together a severity level value and, optionally, a facility. If you do not specify a facility value, the current default facility value specified with the `openlog` function is used.

The severity level value indicates the type of condition this message is associated with and can be one of the following:

#### LOG\_EMERG

An emergency condition; that is, the system cannot be used. This is normally broadcast to all processes.

#### LOG\_ALERT

A condition that must be corrected immediately, such as a corrupted system database.

#### LOG\_CRIT

A critical condition, such as hard device error.

#### LOG\_ERR

An error message.

#### LOG\_WARNING

A warning message.

#### LOG\_NOTICE

A condition that is not an error condition, but may require special handling.

#### LOG\_INFO

An informational message.

#### LOG\_DEBUG

A message that contains information normally used only when debugging a program.

The facility indicates the application or system component generating the message, and can be one of the following:

**Note:** The TPF system does not have a server for all of these facilities; however, the syslog daemon will accept messages if your environment has such a server.

#### LOG\_AUTH

The message is generated by authorization programs.

#### LOG\_DAEMON

The message is generated by system server processes.

## syslog

### **LOG\_LOCAL0**

Reserved for a user-defined facility.

### **LOG\_LOCAL1**

Reserved for a user-defined facility.

### **LOG\_LOCAL2**

Reserved for a user-defined facility.

### **LOG\_LOCAL3**

Reserved for a user-defined facility.

### **LOG\_LOCAL4**

Reserved for a user-defined facility.

### **LOG\_LOCAL5**

Reserved for a user-defined facility.

### **LOG\_LOCAL6**

Reserved for a user-defined facility.

### **LOG\_LOCAL7**

Reserved for a user-defined facility.

### **LOG\_MAIL**

The message is generated by a mail system.

### **LOG\_NEWS**

The message is generated by a news system.

### **LOG\_SYSLOG**

The message is generated by the syslog daemon.

### **LOG\_USER**

The message is generated by random processes. This is the default facility identifier if you do not specify a value.

### **message**

A pointer to the message that is to be sent to the logging facility. This message can be followed by additional arguments in the same way as arguments are specified for the `printf` function. See “`fprintf`, `printf`, `sprintf`—Format and Write Data” on page 201 for more information about the format of these arguments.

#### **Notes:**

1. Occurrences of `%m` in the format string pointed to by the **message** argument are replaced by the error message string associated with the current value of `errno`. A trailing new-line character is added if needed.
2. The total length of the format string and the parameters cannot be greater than 1536 bytes.

## Normal Return

Void.

## Error Return

Not applicable.

## Programming Considerations

- The logged message includes a message header and a message body. The message header consists of a facility indicator, a severity indicator, a time stamp,

a tag string and, optionally, the process ID. The message body is generated from the **message** and any following arguments.

- This function is implemented in dynamic link library (DLL) CTXO. You must use the definition side-deck for DLL CTXO to link-edit an application that uses this function.

## Examples

The following example opens a log facility, sends log messages to the syslog daemon, and closes the facility.

```
#include <syslog.h>

:
/* Open a log facility with the name of local0. Prefix each line in the log file
   with the program name (tpf) and the process ID. */
openlog("tpf", LOG_PID, LOG_LOCAL0);

/* Log an informational message with the specified content */
syslog(LOG_INFO, "Hello from tpf");

/* Close the log facility name. */
closelog();
```

The following is an example of the line produced in the log file by the previous example:

```
<May 26 11:27:51>tpf[3014660]: Hello from tpf
```

## Related Information

- “closelog—Close the System Control Log” on page 48
- “fprintf, printf, sprintf—Format and Write Data” on page 201
- “openlog—Open the System Control Log” on page 386.

## systc–Test System Generation Options

This function is used for the dynamic determination of system generation options at execution time, thereby allowing the code to become object-shippable.

### Format

```
#include <c$systc.h>
int      systc(int option, enum t_systc_ss subsystem);
```

#### option

A valid system-defined generation option defined within the `c$systc.h` header file, or a valid user-defined system generation option defined within the `c$sysug.h` header file.

#### subsystem

##### SYSTC\_BSS

test the specified option in the basic subsystem (BSS).

##### SYSTC\_SS

test the specified option in the subsystem that the ECB is running in.

### Normal Return

- 0 The system or subsystem option bit is off.
- 1 The system or subsystem option bit is on.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example checks the shared DASD feature. If this feature has been generated, an ECB is dispatched for program CELH, passing the string ACT as input to the program.

```
#include <c$systc.h>
char *parmstring = "ACT"
:
:
if (systc(SBSDPS,SYSTC_BSS))
    cremc(strlen(parmstring),parmstring,CELH);
:
:
```

### Related Information

None.

## system—Execute a Command

This function lets you run a TPF program segment under a new entry control block (ECB), wait for the program to be completed, and receive its exit status. You can also specify a standard input/output (I/O) stream using the `system` function. The ECB that calls the `system` function is also called the *parent process*; the new ECB that the `system` function creates is called the *child process*.

The child process runs in the same subsystem and on the same i-stream as the parent process when the parent process calls the `system` function.

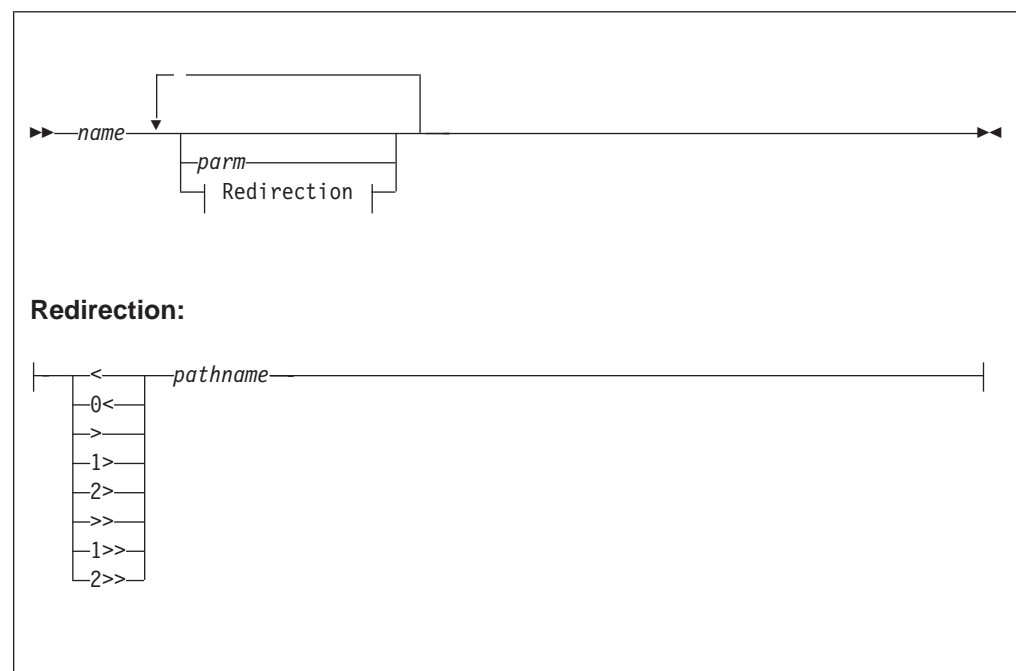
## Format

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

### string

A specification of the program to be run, its parameters, and the files to be opened as standard streams.

The **string** argument has the following format:



Where:

### name

The 4-character name of the TPF program segment to be run. The segment will be run in the same subsystem as the ECB at the time it calls the `system` function.

### parm

A parameter string to be passed to **name**. Each parameter is delimited by 1 or more space characters.

### < or 0<

Indicates redirection of the `stdin` stream.

## system

### > or 1>

Indicates redirection of the stdout stream. If the file exists, it is truncated to zero bytes.

### 2>

Indicates redirection of the stderr stream. If the file exists, it is truncated to zero bytes.

### >> or 1>>

Indicates redirection of the stdout stream without truncation and with output appended to the end of the file.

### 2>>

Indicates redirection of the stderr stream without truncation and with output appended to the end of the file.

### pathname

The name of the file from or to which the stream will be redirected.

The `system` function formats the string argument by removing and processing the redirections, and placing pointers to the **name** and **parm** strings into an argv array so that:

- argv[0] contains the address of the **name** string.
- argv[1] to argv[n] contain the addresses of any **parm** strings, where **n** is the number of **parm** strings specified in the **string** argument passed to the `system` function.
- argv[n+1] contains a null pointer.

The `system` function then builds a parameter list that it passes to the entry point of the **name** program. This parameter list contains two parameters:

- An int parameter containing the total number of strings in the argv array (**n+1**).
- The address of the argv array.

## Normal Return

If the argument is a null pointer, the `system` function returns a value of 1. Otherwise, the `system` function returns the exit status that is returned when the child process exits. Table 16 on page 529 lists the exit status for various ways that the child process can exit.

## Error Return

If the `system` function detects an error in either the input **string** or in the running of the child process, it returns a value of -1 and sets `errno` to one of the following values:

**EINVAL**      The **string** parameter does not contain any nonblank characters (**string** must contain at least a program name) or it contains a vertical bar (|).

**Note:** The TPF system reserves the use of vertical bars for possible future implementation of pipes between processes.

### ETPFYSERR

The child process exited because of a system error.

### E2BIG

The **string** parameter is longer than 4094 bytes.

Table 16. Ways of Exiting a TPF Process and the Resulting Exit Status

How the Child Process Exits	Exit Status Returned by system to Parent Process
return from the initial call to the main function.	The returned value of <i>rc</i> (the return code)
exit( <i>status</i> ) function	The value of <i>status</i>
BAL EXITC RC=Rx	The value contained in Rx
BAL EXITC (with no RC parameter)	Indeterminate
abort()	-1
Any system error that causes the ECB to exit	-1, errno set to ETPFSYSERR

## Programming Considerations

None.

## Examples

The following example creates a new ECB that runs segment QRST. In the main function in QRST, argc will contain the value 4, argv[0] to argv[3] will contain pointers to the strings "QRST", "one", "two", and "three", stdin will read from "my.input", stdout will write to "my.output", and stderr will append to "error.log".

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(void)
{
    errno = 0;
    if (system("QRST one two three "
               "<my.input >my.output 2>>error.log") == -1 &&
        errno != 0)
    {
        printf("system() error:  %s.\n", strerror(errno));
        exit(8);
    }
    return 0;
}
```

## Related Information

- “exit—Exit an ECB” on page 115
- “abort—Terminate Program Abnormally” on page 7.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## tancc–Transaction Anchor Table Control

This function provides the interface to access the transaction anchor table (TANC). It permits an application program to add, remove, or locate a transaction manager control record (TMCR) from the transaction anchor table.

### Format

```
#include <tpfapi.h>
int tancc(enum t_tancc action, TMCR **, GTRID *);
```

#### action

The type of transaction anchor table control (TANCC) requested. This argument must belong to the enumeration type `t_tancc`, which is defined in the `c$tmcr.h` header file.

#### TANCC\_ADD

Adds the pointer of a transaction manager control record (TMCR) to the transaction anchor table (TANC).

#### TANCC\_DELETE

Removes the pointer to a TMCR from the TANC.

#### TANCC\_LOCATE

Locates the pointer to a TMCR with the matching transaction ID (TID).

#### TMCR

A pointer to a TMCR. For a `TANCC_ADD` or `TANCC_DELETE` request, it is the input parameter. For a `TANCC_LOCATE` request, it is the output parameter.

#### GTRID

A pointer to a global transaction ID. This pointer is used only for the `TANCC_LOCATE` request.

### Normal Return

Table 17. *tancc* Normal Return

Value Name	Return Code	Description
TANCC_OK	0	The function is completed successfully.

### Error Return

Table 18. *tancc* Error Return

Value Name	Return Code	Description
TANCC_INVALID_ACTION	-1	The action parameter is incorrect.
TANCC_INVALID_TMCR	-2	The required TMCR parameter is null.
TANCC_INVALID_TID	-3	The required transaction ID (TID) parameter is null.
TANCC_DUPLICATE_TID	-4	A TMCR with the same TID is found in the TANC during a <b>TANCC_ADD</b> process.
TANCC_TID_NOT_FOUND	-5	A TMCR with a corresponding TID was not found during the <b>TANCC_DELETE</b> or <b>TANCC_LOCATE</b> process.



## Programming Considerations

- The calling program needs to have the correct storage protection key to write to the TMCR parameter when using the **TANCC\_LOCATE** parameter.

In the case of a log takeover, there will be more than one transaction anchor table in the system. The tancc function will use a processor index value that is kept in the TMCR (for **TANCC\_ADD** or **TANCC\_DELETE**) or in the entry control block (ECB) (for **TANCC\_LOCATE**) to perform the function on the correct transaction anchor table.

## Examples

The following example shows a normal commit scope.

```
#include <tpfapi.h>
:
{
    TMCR *tmcr;
    GTRID *tid;

    if ( tancc(TANCC_LOCATE, &tmcr, tid) == TANCC_OK)
    {
        /* TMCR is located and pointer is stored in tmcr */
        :
    }
    else {
        :
    }
}
```

## Related Information

None.

## tape\_access—Access a Tape

This function is used to permit an entry control block (ECB) to gain control of a tape without first checking to see if the tape is already open. It also allows the ECB to specify a timeout value if the tape is not immediately available.

### Format

```
#include <tpftape.h>
long tape_access (const char *name,
                  int      io,
                  int      buffmode,
                  char *message,
                  int      timeout);
```

#### name

A pointer to type char, which must be a 3-character string identifying the tape that will be opened. This function can only be called for a general tape.

**io** Code one of the following two terms as defined in the tpftape.h header file.

#### INPUT

To read the tape.

#### OUTPUT

To write to the tape.

#### buffmode

Indicates if the buffered mode of operation will be used. This argument is ignored for input tapes.

#### NOBUFF

A defined term used to specify no buffering (write immediate mode).

#### BUFFERED

A defined term used to specify buffered write mode (preferred).

#### message

A pointer to a message string that will be sent to the console if the function is canceled because of a timeout condition. If a NULL pointer is passed, no message will be sent.

#### timeout

The number of seconds that will lapse before the tape\_access attempt is canceled. If no value is specified, the function will never time out.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a negative return code.

### Programming Considerations

- When completed successfully, the tape will be assigned to the caller and must either be closed or reserved before the ECB exits.
- The tape specified must be a general tape, **not** a real-time tape. Real-time tapes have RT as the first 2 characters of their name.
- If the tape is not physically ready or logically mounted, appropriate messages are sent to the prime computer room agent set (CRAS) for operator intervention.

Control is not returned to the operational program until the tape is ready to be processed or the timeout value has elapsed.

## Examples

The following example uses `tape_access` to gain exclusive control of a tape device.

```
#include <tpftape.h>
long example()
{
    char *tape_name;
    char *message;
    long timeout;
    long tape_return_code;

    :

    if ((tape_return_code = tape_access(tape_name,
                                      INPUT,
                                      BUFFERED,
                                      message,
                                      timeout)) != tape_OK )
    {
        if (tape_return_code == tape_openTimeOut)
        {
            printf("tape_access timed out\n");
        }
        else
        {
            if (tape_return_code == tape_wrongIOmode)
            {
                printf("tape_access failed due to mode conflict\n");
            }
            else
            {
                printf("tape_access failed\n");
            }
        }
    }
    else
    {
        printf("tape_access successful\n");
    }

    :
}
```

## Related Information

- “exit—Exit an ECB” on page 115
- “tape\_close—Close a General Tape” on page 534
- “trsvc—Reserve General Tape” on page 644.

---

**tape\_close—Close a General Tape**

This function closes a general tape, forcing a logical dismount from the system and causing the tape to be physically rewound and unloaded from the tape drive. The named tape must be physically mounted, logically open, and not currently assigned to the issuing ECB.

**Format**

```
#include <tpftape.h>
void      tape_close(const char *name);
```

**name**

This argument is a pointer to type `char`, which must be a 3-character string identifying the tape to be closed. This function can only be called for a general tape.

**Normal Return**

Void.

**Error Return**

Not applicable.

**Programming Considerations**

- The named tape must not be a real-time tape.
- The named tape must be open and mounted, and must NOT be assigned to the ECB at function call time.
- Following closure, the tape is unavailable to the operational program.
- The function ensures that all data from the tape's buffers (control unit buffer and/or host buffer) is physically written to tape before performing the closure.

**Examples**

The following example closes a tape with name VPH.

```
#include <tpftape.h>
:
:
tape_close("VPH");
```

**Related Information**

“[tape\\_open—Open a General Tape](#)” on page 537.

## tape\_cntl—Tape Control

This function performs one of the four above commands related to tape positioning and tape buffer control.

### Format

```
#include <tpftape.h>
int      tape_cntl(const char *name, enum t_cntl command, ...);
```

#### name

This argument is a pointer to type `char`, which must be a 3-character string identifying the tape to be acted upon. This function can only be called for a general tape.

#### command

The action to be performed on the specified general tape. The argument must belong to the enumeration type `t_cntl`, defined in `tpftape.h`. Specify one of the following:

##### CNTL\_FSB

Forward space block. This causes the tape to be moved in the forward direction a specified number of physical blocks. Note that for a blocked tape, a single physical block may represent more than one logical record.

##### CNTL\_FSR

Forward Space Record. This causes the tape to be forward spaced by a specified number of records (blocks).

Two additional parameters are required for Forward Space Block. The first is a valid data level from the enumeration type `t_1vl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The FARW on this data level is modified by the **CNTL\_FSB** subfunction.

The second parameter must be an integer specifying the number of physical blocks to be forward spaced.

##### CNTL\_BSB

Backward space block. This causes the tape to be moved in the backward direction a specified number of physical blocks. Note that for a blocked tape, a single physical block may represent more than one logical record.

Two additional parameters are required for Backward Space Block. The first is a valid data level from the enumeration type `t_1vl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The FARW on this data level is modified by the **CNTL\_BSB** subfunction.

The second parameter is an integer specifying the number of physical blocks to be backward spaced.

##### CNTL\_REW

Rewind. This causes the tape to be positioned at the first record of the current volume, or optionally, at the first record on the first volume in a multi-volume tape data set.

Code the predefined term **FALLBACK** to rewind to the first record on the first volume of a multi-volume data set, or code the defined term **NO\_FALLBACK** to rewind to the first record on the currently mounted volume. Either value may be coded on a single volume data set.

## tape\_cntl

### CNTL\_FLUSH

Flush tape buffer(s). For output tapes, this command causes any write data in the tape buffers (the control unit buffer and/or host buffer) to be written to the tape.

This function requires between two and four arguments, which depend on the term coded for command.

## Normal Return

Integer value of zero.

## Error Return

Nonzero integer contents of CE1SUG.

## Programming Considerations

- The specified tape must have been opened using `tape_open` and must NOT be currently assigned to the issuing ECB. On completion of this function, the tape is left in the *reserved* state, making it available for use by any ECB.
- The specified tape must be currently mounted and ready.
- This function calls the equivalent of `waitc`. To ensure consistent results it is recommended that an explicit `waitc` function be coded prior to the call to `tape_cntl` to ensure that errors for other I/O operations are not reflected in the function return value.
- The **CNTL\_FLUSH** command has no effect on tapes opened for input. If any hardware errors (or end of volume) occur during the buffer flush operation, an automatic switch to the next volume occurs.
- The **FALLBACK/NO\_FALLBACK** options have no meaning for single-volume tape data sets.

## Examples

The following example flushes the write buffer, rewinds, and positions the VPH tape (already opened) at the fourth physical block. The program checks that the **CNTL\_FSB** request completed successfully before continuing.

```
#include <tpftape.h>
:
:
tape_cntl("VPH", CNTL_FLUSH);
tape_cntl("VPH", CNTL_REW, FALLBACK);
if (tape_cntl("VPH", CNTL_FSB, D6, 3))
{
    serrc_op(SERRC_EXIT,0x1234,"ERROR ON VPH TAPE",NULL) ;
}
:
:
:
```

## Related Information

- “`tape_close`—Close a General Tape” on page 534
- “`tape_open`—Open a General Tape” on page 537.

## tape\_open—Open a General Tape

This function makes a general tape available to the operational program.

### Format

```
#include <tpftape.h>
void      tape_open(const char *name, int io);
```

#### name

This argument is a pointer to type `char`, which must be a 3-character string identifying the tape to be opened. This function can only be called for a general tape.

**io** Code 1 of these 2 terms as defined in header file `tpftape.h` **INPUT** to read the tape, or **OUTPUT** to write to the tape.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The tape specified by the **name** argument must not be open at the time of the function call.
- The specified tape must be defined to the system.  
The specified tape is positioned at the first record and is in the *reserved* state, meaning that it is available for use by any ECB.
- The tape specified must be a general tape, *not* a real-time tape. Real-time tapes have RT as the first 2 characters of their name.
- If the tape is not physically ready or logically mounted, appropriate messages are sent to the prime CRAS for operator intervention. Control is not returned to the operational program until the tape is ready to be processed.

## Examples

The following example opens a VPH tape for output and issues a system error with message if the tape is not available for use.

```
#include <tpftape.h>
:
:
tape_open("VPH", OUTPUT);
```

## Related Information

"tape\_close—Close a General Tape" on page 534.

## tape\_read—Read a Record From General Tape

This function reads a record from a general tape.

### Format

```
#include <tpftape.h>
void *tape_read(const char *name, enum t_lvl level,
                enum t_blktype size);
```

#### name

This argument is a pointer to type `char`, which must be a three-character string identifying the tape to be read from. This function can only be called for a general tape.

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0-F). This argument identifies the CBRW on which the working storage block containing the tape record image will be placed.

#### size

One of 4 possible values (L0, L1, L2, L4) from the enumeration type `t_blktype`, representing the working storage block size of the record on tape:

**L0** 127 bytes

**L1** 381 bytes

**L2** 1055 bytes

**L4** 4095 bytes.

### Normal Return

Pointer to the working storage block containing the tape record image.

### Error Return

NULL.

### Programming Considerations

- The specified tape must have been opened using `tape_open`, and must NOT be currently assigned to the issuing ECB. On completion of this function, the tape is left in the *reserved* state, meaning it is available for use by any ECB.
- This function calls the equivalent of `waitc`. As a result, all pending I/O occurs before control is returned to the calling program.
- The CBRW level specified must be unoccupied when the function is called.
- On return from the function, the specified tape has been placed in the *reserved* state for use by other ECBs.
- If the tape is mounted in blocked mode, the application should not assume that the execution of this function causes either any physical I/O to be initiated or the application timeout value to be reset.

### Examples

The following example opens, reads a record from, and closes a VPH tape mounted for input.



```

#include <tpftape.h>
struct vp0vp *vp;
:
tape_open("VPH",INPUT);

if((vp = (struct vp0vp *) tape_read("VPH",D6,L2)) == NULL)
    exit(0x56789);          /* Dump & exit if read failed */
tape_close("VPH");          /* Close it. */

```

## Related Information

- “tape\_close—Close a General Tape” on page 534
- “tape\_open—Open a General Tape” on page 537
- “tape\_write—Write a Record to General Tape” on page 540.

## tape\_write—Write a Record to General Tape

This function writes a record to a general tape.

### Format

```
#include <tpftape.h>
void      tape_write(const char *name, enum t_lvl level);
```

#### name

This argument is a pointer to type `char`, which must be a three-character string identifying the tape to be written to. This function can only be called for a general tape.

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This argument identifies the CBRW containing the record to be written to tape.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The specified tape must have been opened using `tape_open`, and must NOT be currently assigned to the issuing ECB. On completion of this function, the tape is left in the *reserved* state, making it available for use by any ECB.
- The status of the I/O operation can never be determined by the operational program.
- The working storage block resident on the CBRW level specified as argument **level** is released and is no longer available to the operational program.
- The FARW on the level indicated by argument **level** is not altered.

## Examples

The following example opens and writes a copy of the record on level D1 to the VPH tape.

```
#include <tpftape.h>
:
:
tape_open("VPH", OUTPUT);
tape_write("VPH", D1);    /* Write block on D1 to tape    */
```

## Related Information

- “`tape_close`—Close a General Tape” on page 534
- “`tape_open`—Open a General Tape” on page 537
- “`tape_read`—Read a Record From General Tape” on page 538.

## tasnc—Assign General Tape to ECB

This function assigns the specified general tape name to the issuing ECB, permitting the issuing ECB to have exclusive I/O access to the tape.

### Format

```
#include <tpftape.h>
void      tasnc(const char *name);
```

#### name

This argument is a pointer to type char, which must be a three-character string identifying the tape to be assigned. This function can only be called for a general tape.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- This function is required before issuing any tape operations if trsvc has been called.
- If the specified tape is not currently in the reserved state (that is, it is assigned to another ECB), processing of the current ECB is suspended until the tape becomes available.

## Examples

The following example assigns tape VPH to the ECB that is executing now.

```
#include <tpftape.h>
:
:
tasnc("VPH");
```

## Related Information

- “tbspc—Backspace General Tape and Wait” on page 542
- “tape\_close—Close a General Tape” on page 534
- “topnc—Open a General Tape” on page 556
- “tprdc—Read General Tape Record” on page 641
- “trewc—Rewind General Tape and Wait” on page 643
- “trsvc—Reserve General Tape” on page 644
- “twrtc—Write a Record to General Tape” on page 646.

## tbspc—Backspace General Tape and Wait

This function backspaces a general tape a specified number of records and waits until the tape is correctly positioned before returning control to the user. If this function is issued to a tape that is mounted in blocked mode, a system error will occur.

### Format

```
#include <tpftape.h>
int tbspc(const char *name, enum t_lvl level, int fallback);
```

#### name

This argument is a pointer to type `char`, which must be a 3-character string identifying the tape to be backspaced. This function can only be called for a general tape.

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter specifies the FARW from which the number of records to backspace is to be obtained, right justified in CE1F`x`.

#### fallback

This argument applies when operating on a multivolume tape file. If **FALLBACK** is specified, the previous volume is mounted if the load point was reached on the current volume and more records exist to backspace. If **NO\_FALLBACK** is specified, backspacing stops at the load point with no fallback to the previous volume.

### Normal Return

Integer value of zero if the operation is successful.

### Error Return

The value of CE1SD`x` is returned if an I/O error occurs.

## Programming Considerations

- Specifying an invalid data level results in a system error with exit.
- The tape must already have been assigned to the issuing ECB before this function is called.
- This function calls the equivalent of `waitc`.
- The number of physical blocks to be backspaced must be from 1 to 65 535. Attempts to backspace more than 65 535 records will backspace the number of records equal to the value entered mod 64 K.

## Examples

The following example backspaces the VPH tape by the specified number of records.

```
#include <tpftape.h>
int recs = 3;
:
:
*((short *)&ecbptr()->celfh9)) = recs;
if (tbspc("VPH",D9,FALLBACK))
{
    serrc_op(SERRC_EXIT,0x1234,"ERROR BACKSPACING VPH TAPE",NULL) ;
}
```

## **Related Information**

- “`tape_close`—Close a General Tape” on page 534
- “`waitc`—Wait For Outstanding I/O Completion” on page 686
- “`tasnc`—Assign General Tape to ECB” on page 541.

---

## tclsc—Close a General Tape

This function closes a general tape, forcing a logical dismount from the system and causing the tape to be physically rewound and unloaded from the tape drive. The named tape must be physically mounted, logically open, and assigned to the issuing entry control block (ECB).

### Format

```
#include <tpftape.h>
void tclsc(const char *name);
```

#### name

This argument is a pointer to type char, which must be a 3-character string identifying the tape to be closed. This function can only be called for a general tape.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The named tape must not be a real-time tape, and it must be opened and mounted.
- Following closure, the tape is unavailable to the operational program.
- The function calls the equivalent of `tsync` before closing the tape.

## Examples

The following example closes a tape with name VPH.

```
#include <tpftape.h>
:
:
tclsc("VPH");
```

## Related Information

“`tape_open`—Open a General Tape” on page 537.

## tdspc–Display Tape Status

This function returns the status indicators and device address for active or standby tapes.

### Format

```
#include <tpftape.h>
struct tpstat *tdspc(const char *name, char type, enum t_lvl level);
```

#### name

This argument is a pointer to type char, which must be a 3-character string identifying the tape whose status is to be obtained.

#### type

Whether the active or standby tape status is to be checked. Use **ACTIVE** to specify the active tape, or **STANDBY** to specify the standby tape.

#### level

One of 16 possible values representing a valid data level from the enumeration type t\_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). This parameter identifies the FARW on which the status information is to be placed.

### Normal Return

Pointer to structure tpstat (defined in tpftape.h) containing tape status. The indicated tape need not be assigned to the issuing ECB.

### Error Return

NULL.

### Programming Considerations

Real-time tape names can be given aliases, or tape pseudonyms. These pseudonyms are defined using RTMAP definitions in CEFZ. This mechanism provides a means for mapping many names to just a few tape devices.

### Examples

The following example calls tdspc to examine the status of the active VPH tape.

```
#include <tpftape.h>
struct tpstat *status;
:
:
if ((status = tdspc("VPH", ACTIVE, D0)) == NULL)
{
    serrc_op(SERRC_EXIT,0x1234,"VPH TAPE NOT MOUNTED",NULL) ;
}
```

### Related Information

None.

---

## tdspc\_q–Display Tape Queue Length

This function returns the module queue length of the specified active tape.

### Format

```
#include <tpftape.h>
struct tpqstat *tdspc_q(const char *name, enum t_lvl level);
```

#### name

This argument is a pointer to type char, which must be a 3-character string identifying the tape whose queue length is to be obtained.

#### level

This argument identifies the FARW on which the queue length information is to be placed. It can be one of 16 possible values representing a valid data level from the enumeration type t\_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F).

### Normal Return

Pointer to structure tpqstat (defined in tpftape.h) containing the tape queue length. The indicated tape need not be assigned to the issuing ECB.

### Error Return

NULL.

### Programming Considerations

None.

### Examples

The following example calls the tdspc\_q function to obtain the queue length of the active "VPH" tape.

```
#include <tpftape.h>
struct tpqstat *status;
:
:
status = tdspc_q("VPH", D0);
if (0 == status.q_length)
{
    serrc_op(SERRC_EXIT,0x1234,"VPH TAPE QUEUE EMPTY",NULL) ;
}
```

### Related Information

None.



## tdspc\_v—Retrieve VOLSER for a Specified Tape Name

This function is used to retrieve the tape volume serial number (VOLSER) for the specified tape name.

### Format

```
#include <tpftape.h>
tpvstat *tdspc_v (const char *name,
                  char type,
                  enum t_lvl level);
```

#### name

A pointer to type char, which must be a 3-character string identifying the tape whose status will be obtained.

#### type

Indicates whether the active or standby tape status will be checked.

##### ACTIVE

Used to specify the active tape.

##### STANDBY

Used to specify the standby tape.

#### level

One of 16 possible values representing a valid data level from enumeration type t\_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). This parameter identifies the file address reference word (FARW) on which the status information will be placed.

### Normal Return

When completed successfully, the return code will point to the tpvstat structure that was placed in the FARW of the data level specified.

### Error Return

An error return is indicated by a zero return code.

## Programming Considerations

When returned, the data level will be unchanged, but the FARW will contain the VOLSER and tape module number.

### Examples

The following example retrieves the VOLSER from the tape name indicated.

```
#include <tpftape.h>
long example()
{
    char *tape_name;
    struct tpvstat *tape_volser_status;

    :

    tape_volser_status = tdspc_v (tape_name,ACTIVE,DF);
    printf("the volser is %6s\n",tape_volser_status->vs);

    :

}
```

**tdspc\_v**

## **Related Information**

- “tdspc–Display Tape Status” on page 545
- “tdspc\_q–Display Tape Queue Length” on page 546.

## tdtac—Issue a User-Specified Data Transfer CCW

This function is used to start a single data transfer channel control word (CCW) for the specified tape name.

### Format

```
#include <tpftape.h>
long tdtac (const char *name,
            enum t_lvl level);
```

#### name

A pointer to type `char`, which must be a 3-character string identifying the tape whose status will be obtained.

#### level

One of 16 possible values representing a valid data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter identifies the file address reference word (FARW), which contains a format-1 CCW. On return, the data level will be unchanged.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a negative return code.

The following error codes are common for this function:

- 1 End of tape.
- 2 Hardware error.
- 3 Long length record.
- 4 Short length record.

The details of the error can also be obtained by examining the value in the `CE1SUG` field in the entry control block (ECB).

### Programming Considerations

- This function calls the equivalent of the `waitc` function.
- Command chaining and data chaining are not allowed.
- The FARW for the data level specified by the `level` parameter must contain the format-1 data transfer CCW.
- The general tape specified by the `name` parameter must be assigned to this ECB when this function is called.
- During normal processing of this function, the CCW contained in the FARW is not modified. If an incorrect record length is found, the length field will be in the last 2 bytes of the FARW on the data level specified by the macro.
- This function runs independently of normal tape handling and volume switching facilities. When an unusual condition such as end-of-file, end-of-tape, or hardware error occurs, a return is made to the operational program by way of the `waitc` return mechanism. Note that write operations are allowed to be completed when an end-of-tape condition is detected and the operation is appropriately

## tdtac

flagged to indicate this condition. For this function only, an end-of-tape condition on a write operation is flagged by the same indicator bit as an end-of-file condition on a read operation.

- Do not use this function to write heap storage areas to a blocked tape.
- If an attempt is made to write a record longer than the maximum length, a system error is issued.
  - For write operations to a blocked tape, the maximum byte count of the record being written is 32 752.
  - For write operations to an unblocked tape, the maximum byte count of the record being written is 65 535.
- The minimum record size that can be written or read is 16 bytes.
- For write and sense commands, the suppress length indication flag is automatically set by the control program.
- This function cannot be used to issue a Read Backwards command to a blocked tape. If this is attempted with a blocked tape, a system error is issued.
- No check is made by the control program to see if the command is supported by the device. For example, a command code to read configuration data can be sent to a 3480 tape device even though it is not defined for that device type.

## Examples

The following example retrieves the current tape position using a Read Block ID CCW, which was issued through the tdtac function.

```
#include <tpftape.h>
#include <c$tpxd.h>
long example()
{
    CW0CCW          temp_ccw;
    long            tape_return_code;
    TPFxd_location  *positioningString;

    temp_ccw.cw0ccw1.cw0cmd1 = tape_ccw_rbid;
    temp_ccw.cw0ccw1.cw0bct1 = sizeof(positioningString->offset);
    temp_ccw.cw0ccw1.cw0adr1 = &positioningString->offset;
    temp_ccw.cw0ccw1.cw0flg1 = CW0SLI;
    memcpy(&(ecbptr()->celfaf),&temp_ccw,sizeof(temp_ccw));

    tdtac (token->ext_name,DF);
}
```

## Related Information

- “tape\_access—Access a Tape” on page 532
- “tape\_cntl—Tape Control” on page 535
- “tasnc—Assign General Tape to ECB” on page 541
- “tclsc—Close a General Tape” on page 544
- “topnc—Open a General Tape” on page 556
- “tpcnc—Issue a User-Specified Control Operation CCW” on page 560
- “trsvc—Reserve General Tape” on page 644.

## tmpfile—Create a Temporary File

This function creates a temporary binary file. It opens the temporary file in `wb+` mode. The file is automatically removed when it is closed or when the program ends.

### Format

```
#include <stdio.h>
FILE *tmpfile(void);
```

### Normal Return

If successful, this function returns a pointer to the stream associated with the file that is created.

### Error Return

If the `tmpfile` function cannot open the file, it returns a `NULL` pointer.

The following are the possible values of `errno`:

<b>EMFILE</b>	<code>OPEN_MAX</code> file descriptors are currently open in the calling process.
	<code>FOPEN_MAX</code> streams are currently open in the calling process.
<b>ENFILE</b>	The maximum number of files that are allowed is currently open in the system.
<b>ENOSPC</b>	The file system that will contain the new file cannot be expanded.
<b>ENOMEM</b>	There is not enough storage space available.

## Programming Considerations

Temporary files cannot be created in 1052 or UTIL state.

## Examples

The following example creates a temporary file and, if successful, writes the character string, `tmpstring`, to it. When the program ends, the file is removed.

```
#include <stdio.h>

int main(void) {
    FILE *stream;
    char tmpstring[] = "This string will be written.";

    if((stream = tmpfile( )) == NULL)
        printf("Cannot make a temporary file\n");
    else
        fprintf(stream, "%s", tmpstring);
}
```

## Related Information

“`fopen`—Open a File” on page 199.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## tmpnam—Produce a Temporary Name

This function produces a temporary name.

### Format

```
#include <stdio.h>
char *tmpnam(char *string);
```

#### **string**

The address of an array of characters where the temporary name is to be stored or a null pointer.

This function produces a valid file name that is not the same as the name of any existing file. It stores this name in **string**. If **string** is a NULL pointer, the tmpnam function leaves the result in an internal static buffer. Any subsequent calls may modify this object. If **string** is not a NULL pointer, it must point to an array of at least L\_tmpnam bytes. The value of L\_tmpnam is defined in the stdio.h header file.

### Normal Return

If **string** is a NULL pointer, the tmpnam function returns the pointer to the internal static object in which the generated unique name is placed. Otherwise, if **string** is not a NULL pointer, it returns the value of **string**. The tmpnam function produces a different name each time it is called in a module up to at least TMP\_MAX names. Files created using names returned by tmpnam are not automatically discarded at the end of the program.

### Error Return

Not applicable.

### Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

### Examples

The following example calls the tmpnam function to produce a valid file name.

```
#include <stdio.h>

int main(void)
{
    char *name1;
    if ((name1 = tmpnam(NULL)) != NULL)
        printf("%s can be used as a file name.\n", name1);
    else printf("Cannot create a unique file name\n");
}
```

### Related Information

“fopen—Open a File” on page 199.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## tmslc–ECB Time Slice Facility

This function is used to enable or disable time slicing for an ECB.

When an ECB is enabled for time slicing, the ECB will lose control at defined time intervals. This allows other tasks in the system to receive control.

### Format

```
#include <sysapi.h>
int tmslc(int flags, char *name);
```

#### flags

The following flags are defined in `sysapi.h`:

#### **TMSLC\_ENABLE or TMSLC\_DISABLE**

Choose only one of the previous options.

##### **TMSLC\_ENABLE**

Allow the ECB to time slice.

When the entry control block (ECB) reaches the `RUNTIME` value, it gets suspended. The ECB will remain suspended for the value specified in the `MINSUSP` parameter.

##### **Notes:**

1. `RUNTIME` and `MINSUSP` are values defined with the **name** parameter.
2. If you do not specify **TMSLC\_HOLD** and the ECB is holding a resource, the ECB will not get suspended.

##### **TMSLC\_DISABLE**

Disable time slicing.

Return the ECB to normal operation.

##### **TMSLC\_HOLD**

This optional parameter is valid only for the **TMSLC\_ENABLE** option. This optional parameter indicates if the ECB can be suspended while holding a resource (`fiwhc`, `corhc`, `tasnc`, `evnwc`, or `glob_lock`).

**Note:** This does not apply to resources held with the `lockc` function.

##### **TMSLC\_LETRUN**

This optional parameter is valid only for the **TMSLC\_DISABLE** option. If you code **TMSLC\_LETRUN**, the ECB will not lose control. If you do not code **TMSLC\_LETRUN**, the ECB will lose control unless the ECB was not enabled with **TMSLC\_HOLD** and the ECB is currently holding a resource.

#### name

The time-slice **name** parameter is a pointer to a 1- to 8-character name. This **name** must be null-terminated. The time-slice name has four time-slice parameters associated with it: `RUNTIME`, `MAXTIME`, `MINSUSP`, and `MAXECB`.

The predefined time-slice names and values of the parameters associated with them are shown in Table 19.

### Normal Return

The return code will be 0.

## Error Return

A negative return code indicates that an error occurred:

- 1 The name specified for the **name** parameter is not recognized.
- 2 The maximum number of ECBs that can be time sliced is already active for the specified time-slice name.

## Programming Considerations

- This macro cannot be run while the system is in restart.
- Using the `tmslc` function requires authorization to issue a restricted function.
- Do **not** use this function so that time slicing is enabled throughout the life of the transaction because there is no protection against having an ECB that can be time-sliced call a segment that cannot allow time-slicing.
- Only enable time slicing in specific CPU-intensive code paths. Time slicing is not supported across TPF services such as:
  - Calls to TPF real-time segments (CYYM, CYYA, BPKD, and others)
  - TPFDF library functions
  - TPF application programming interface (API) library functions

because these services may reference shared memory, issue `$LOCKC` macros, or hold critical resources. Ensure that time slicing is disabled before calling such system services.

Time slicing can be enabled across standard C library functions that are supported by the TPF 4.1 system and have been identified as not being modified for the TPF system environment.

- An ECB enabled for time slicing must never issue a `$LOCKC` macro.
- An ECB enabled for time slicing must never update a global field or other storage area that can be updated simultaneously by other ECBs.
- Coding **TMSLC\_LETRUN** is advantageous only in a highly repetitive loop where time slicing is enabled and disabled frequently. Otherwise, to avoid timeout errors, do not specify **TMSLC\_LETRUN**.
- An ECB that is enabled for time slicing and that is running with the **TMSLC\_HOLD** parameter can hold a resource while suspended. This can cause hang conditions. Resources held by an ECB that can be time sliced should not be needed elsewhere.
- An ECB that is enabled for time slicing may be exited with a 000010 system error if the **TMSLC\_HOLD** parameter is not specified. This can occur if the ECB is holding a resource (which prevents time slicing) and has been running without giving up control for greater than the time allowed by the application timeout value (500 ms).
- An ECB will exit with a 000010 system error if the ECB is not forced to give up control and the ECB continues to run for 500 ms without giving up control.
- The maximum amount of accumulated run time that an ECB enabled for time slicing is allowed before exiting with a 002010 system error is not reset each time a **TMSLC\_ENABLE** parameter is issued. The amount of time set for the `MAXTIME` value should allow the ECB to complete its task.
- ECBs that are suspended because of a `tmslc` call will be purged during system cycle-down to 1052 state unless they have been previously identified to survive cycle-down.
- Resource control is shipped with three predefined time-slice names. Table 19 shows the predefined time-slice names and values of the parameters associated



with them.

Table 19. Time-Slice Name Table

Time-Slice Name	RUNTIME Value	MAXTIME Value	MINSUSP Value	MAXECB Value
<b>IBMLOPRI</b>	50 ms	20000 ms	1000 ms	50
<b>IBMHIPRI</b>	100 ms	10000 ms	100 ms	50
<b>IBMINDEF</b>	50 ms	0 ms	2000 ms	20

- These are the values for the time-slice name parameters as shipped by IBM.
- IBMLOPRI, IBMHIPRI, and IBMINDEF are reserved for use by IBM.
- To add new time-slice names, use the ZTMSL command or the TMSLC macro with the ASSIGN parameter. You can also use the ZTMSL command to display, change, and remove time-slice names.

## Examples

The following example allows an ECB to be time sliced based on the attributes associated with the time-slice name, BIGSORT. The ECB can be time sliced even if it is holding a resource.

```
#include <sysapi.h>
void QZZ0();

:
rc = tmslc( TMSLC_ENABLE+TMSLC_HOLD, "BIGSORT" );
if (rc < 0)
    /* warn the user that an error occurred */
else
    /* start running in time slice mode */

:
tmslc( TMSLC_DISABLE, NULL );
```

## Related Information

None.

## topnc—Open a General Tape

This function makes the specified general tape available to the currently executing ECB.

### Format

```
#include <tpftape.h>
void topnc(const char *name, int io, int bufmode);
```

#### name

This argument is a pointer to type `char`, which must be a 3-character string identifying the tape to be opened. This function can only be called for a general tape.

**io** This argument is treated as an integer describing whether the tape is to be read (input) or written (output). Use the defined terms **INPUT** to denote an input tape or **OUTPUT** to denote an output tape.

#### bufmode

Indicates if buffered mode of operation is to be used. This argument is ignored for input tapes. Use the defined terms **NOBUFF** to denote no buffering (write immediate mode), or **BUFFERED** to denote buffered write mode (preferred).

### Normal Return

Void. The specified tape is positioned at the first record, and the tape has been assigned to the issuing ECB.

### Error Return

Not applicable.

## Programming Considerations

- The tape specified as the **name** argument must not be open at the time of the function call.
- The tape specified as the **name** argument must be defined to the system.
- The tape specified as the **name** argument must be a general tape, *not* a real-time tape. Real-time tapes have RT as the first 2 characters of their name.
- If the tape is not physically ready or logically mounted, appropriate message(s) are sent to the prime CRAS for operator intervention. Control is not returned to the operational program until the tape is ready to be processed.

## Examples

The following example opens the VPH tape for input in tape write immediate mode.

```
#include <tpftape.h>
:
:
topnc("VPH", INPUT, NOBUFF);
```

## Related Information

- “exit—Exit an ECB” on page 115
- “tape\_close—Close a General Tape” on page 534
- “trsvc—Reserve General Tape” on page 644.

## tourc—Write Real-Time Tape Record/Release Storage Block

This function causes the record on the specified data level to be written to a real-time tape, and returns the block to the available pool.

### Format

```
#include <tpftape.h>
void    tourc(const char *name, enum t_lvl level);
```

#### name

This argument is a pointer to type `char`, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this level is the record to be written to tape.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- Use this function to write a record to a real-time tape when the storage block used for a buffer is no longer needed by the application. If an invalid tape name is passed as an argument, return is made to the caller with no action taken.
- Specifying an invalid data level results in a system error with `exit`.
- The status of the operation can never be determined by the operational program.
- If the tape is mounted in blocked mode, the application should not assume that the execution of this function causes either any physical I/O to be initiated or the application time-out value to be reset.

## Examples

The following example writes the working storage block on level D9 to the RTA tape. On return, the block is no longer available to the operational program.

```
#include <tpftape.h>
:
:
tourc("RTA",D9);
```

## Related Information

- “toutc—Write Real-Time Tape Record” on page 558
- “twrtc—Write a Record to General Tape” on page 646.

## toutc—Write Real-Time Tape Record

This function causes the record on the specified data level to be written to a real-time tape. The first four bytes of the file address reference word (FARW) of the data level specified must contain the address of the record to be written. The last 2 bytes of the FARW must contain the byte count of the record.

The toutc function does not return the block of storage to the appropriate pool but retains it on the specified level following return from a subsequent waitc.

### Format

```
#include <tpftape.h>
void      toutc(const char *name, enum t_lvl level, int bufmode);
```

#### name

This argument is a pointer to type char, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.

#### level

One of 16 possible values representing a valid data level from the enumeration type t\_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). The working storage block on this level is the record to be written to tape.

#### bufmode

Whether the 3480 being written to is written in buffered mode or write immediate mode. Use the defined terms **NOBUFF** to denote no buffering, or **BUFFERED** to denote buffered write mode.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- Use this function to write a record to a real-time tape when the storage block used for a buffer is needed by the application in later processing. To ensure completion of the operation, waitc must be called.
- A real-time tape cannot be assigned. If an invalid tape name is passed as an argument, return is made to the caller with no action taken.
- The maximum data length (that is stored in the last 2 bytes of the FARW at ce1fh9) is 4095.

### Examples

The following example writes the working storage block on level D9 to the RTA tape in buffered mode. A message is issued if an I/O error occurs.

```
#include <tpftape.h>
:
:
ecbptr()->ce1fa9 = (long int) (ecbptr()->ce1cr9); /* get address in FARW */
(*(short int *)&(ecbptr()->ce1fh9)) = 128;      /* length of record */
toutc("RTA",D9,BUFFERED);
```

```
if (waitc())  
{  
    serrc_op(SERRC_EXIT,0x1234,"ERROR ON RTA TAPE",NULL) ;  
}
```

## Related Information

- “tourc—Write Real-Time Tape Record/Release Storage Block” on page 557
- “twrtc—Write a Record to General Tape” on page 646.

## tpcnc—Issue a User-Specified Control Operation CCW

This function is used to start a single control operation channel control word (CCW) for the specified tape name.

### Format

```
#include <tpftape.h>
long tpcnc (const char *name,
            enum t_lvl level);
```

#### name

A pointer to type `char`, which must be a 3-character string identifying the tape whose status will be obtained.

#### level

One of 16 possible values representing a valid data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This parameter identifies the file address reference word (FARW), which contains a format-1 CCW. On return, the data level will be unchanged.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero return code. The details of the error can be obtained by examining the value in the `CE1SUG` field in the entry control block (ECB).

## Programming Considerations

- This function calls the equivalent of the `waitc` function.
- Command chaining and data chaining are not allowed.
- The FARW for the data level specified by the **level** parameter must contain the format-1 data transfer CCW.
- The general tape specified by the `name` parameter must be assigned to this ECB when this function is called.
- During normal processing of this function, the CCW contained in the FARW is not modified. If an incorrect record length is found, however, the byte count in the CCW will be adjusted to reflect the actual number of bytes transferred.
- This function runs independently of normal tape handling and volume switching facilities. When an unusual condition such as end-of-file, end-of-tape, or hardware error occurs, a return is made to the operational program by using the `waitc` return mechanism.
- No check is made by the control program to see if the command is supported by the device.

## Examples

The following example writes a tapemark using the `tpcnc` function.

```
#include <tpftape.h>
long example()
{
    CW0CCW          temp_ccw;
    long             ioreturncode;

    temp_ccw.cw0ccw1.cw0cmd1 = tape_ccw_wtm;
```

```

temp_ccw.cw0ccw1.cw0bct1 = 1;
temp_ccw.cw0ccw1.cw0adr1 = 0;
temp_ccw.cw0ccw1.cw0flg1 = 0;
memcpy(&(ecbptr()->celfaf),&temp_ccw,sizeof(temp_ccw));

ioreturncode = tpcnc (token->ext_name,DF);
}

```

## Related Information

- “tape\_access—Access a Tape” on page 532
- “tape\_cntl—Tape Control” on page 535
- “tasnc—Assign General Tape to ECB” on page 541
- “tclsc—Close a General Tape” on page 544
- “topnc—Open a General Tape” on page 556
- “trsvc—Reserve General Tape” on page 644.

## TPF\_CALL\_BY\_NAME, TPF\_CALL\_BY\_NAME\_STUB–Call (Enter) a Program by Name

The TPF\_CALL\_BY\_NAME\_STUB() macro defines a call stub variable, which can be passed to a TPF\_CALL\_BY\_NAME() macro. The TPF\_CALL\_BY\_NAME() macro returns a pointer to a function value which can be used to transfer control to a TPF segment. Calling the function pointer executes enter with return linkage to the specified **program**. The current active program remains held by the entry control block (ECB). The address of the next sequential instruction (NSI) in the current program is saved for an expected return.

### Format

```
#include <tpfapi.h>
TPF_CALL_BY_NAME_STUB(stubname);
funtype TPF_CALL_BY_NAME(const char *program, stubname, funtype);
```

#### stubname

The ordinary identifier name of the call stub to be defined by the TPF\_CALL\_BY\_NAME\_STUB() macro and passed to the TPF\_CALL\_BY\_NAME() macro.

#### funtype

A *pointer to function* type of the same type as the address of the entry point of the application program you want to enter. If the application program is a TPF BAL segment, **funtype** must be a pointer to a function that takes a single parameter of type (struct TPF\_regs \*) and returns a pointer, integer, or void type. The tpfapi.h header file provides 2 typedef identifiers for declaring functions and pointers to functions of this type, TPF\_BAL\_FN and TPF\_BAL\_FN\_PTR, defined as:

```
typedef void TPF_BAL_FN(struct TPF_regs *);
typedef TPF_BAL_FN *TPF_BAL_FN_PTR;
```

#### program

A pointer to the name of the application program you want to enter. **program** must be the 4-character name of a BAL segment, a TARGET(TPF) segment, or a TPF ISO-C DLM.

### Normal Return

The TPF\_CALL\_BY\_NAME() macro returns a pointer to function type as specified by the **funtype** parameter.

### Error Return

Not applicable.

### Programming Considerations

- These macros cannot be called from either of the following:
  - C++ language
  - C language that is compiled using the dynamic link library (DLL) option.
- The TPF\_CALL\_BY\_NAME\_STUB() macro should be used only to define stub variables that will be passed to the TPF\_CALL\_BY\_NAME() macro.
- The **stubname** parameter that is passed to each TPF\_CALL\_BY\_NAME() macro must have previously been defined by a call to the TPF\_CALL\_BY\_NAME\_STUB() macro.
- There are 3 parts to calling a program by name using the TPF\_CALL\_BY\_NAME() macro:



## TPF\_CALL\_BY\_NAME, TPF\_CALL\_BY\_NAME\_STUB

1. Declaring a typedef for the type of the address of the entry point in the called program (**funtype**), or, in other words, a pointer to function type.
  2. *Defining* a call stub variable by calling the TPF\_CALL\_BY\_NAME\_STUB() macro. This macro must be called in the declaration section of a block (that is, before any statements) or at file scope. If it is called at block scope, it defines the stub in automatic storage. If it is called at file scope, it defines the stub with external linkage in writable static storage; therefore, the source file in which it is called must be compiled with the RENT option. The same **stubname** can be reused for more than one TPF\_CALL\_BY\_NAME() macro call.
  3. Calling the TPF\_CALL\_BY\_NAME() macro, passing it the program name, stub variable identifier (**stubname**) defined by the previous TPF\_CALL\_BY\_NAME\_STUB() macro, and program entry point address type (**funtype**) declared by the previous typedef. The TPF\_CALL\_BY\_NAME() macro returns a function pointer value of type **funtype**.
- Each execution of TPF\_CALL\_BY\_NAME() modifies the stub variable **stubname** passed to it.
  - Subsequent calls to TPF\_CALL\_BY\_NAME() may invalidate the pointer to function value returned by a previous call to TPF\_CALL\_BY\_NAME().

## Examples

The following example shows the 3 steps for entering a TPF program by name using the TPF\_CALL\_BY\_NAME() macro.

```
#include <tpfapi.h>

/*****
/* 1. Declare (typedef) function pointer types for the programs you
/* need to enter (this is often done in a header file which can be
/* included in multiple applications).
*****/
typedef char *(*chptrfn)(int);
typedef void (*voidfn)(void);

char *XYZ0(int index)
{
    /*****
    /* 2. Call the TPF_CALL_BY_NAME_STUB() macro to define a stub
    /* variable which can be passed to the TPF_CALL_BY_NAME()
    /* macro.
    *****/
    TPF_CALL_BY_NAME_STUB(cbn_stub);

    /*****
    /* 3a. Call the TPF_CALL_BY_NAME() macro to execute an enter by
    /* name with return. This statement calls a (char (*)(int))
    /* type function.
    *****/
    char *program = TPF_CALL_BY_NAME("ABCD", cbn_stub, chptrfn)(index);

    /*****
    /* 3b. This statement calls a (void (*)(void)) type function.
    *****/
    TPF_CALL_BY_NAME(program, cbn_stub, voidfn)();

    return program;
}
```

## Related Information

- “entdc, \_\_ENTDC—Enter a Program and Drop Previous Programs” on page 104
- “entrc—Enter a Program with Expected Return” on page 106.

## tpf\_cfconc—Connect to a Coupling Facility List or Cache Structure

This function allocates a coupling facility (CF) list or cache structure and connects to it, or connects to a CF list or cache structure that is allocated already. A *CF list* or *cache structure* is a named piece of storage on a CF.

### Format

```
#include <c$fapi.h>
int tpf_cfconc(struct icfco *parms);
```

#### parms

A pointer to the tpf\_cfconc parameter list.

### Normal Return

#### ICF\_SUCCESS

The function is completed successfully and the TPF system returns data in a tpf\_cfconc answer area, which is a structure of type ICFAA. The c\$cfaa.h header file contains ICFAA.

### Error Return

One of the following:

#### ICFRRCBADPARMLIST

A program error occurred because a pointer to the tpf\_cfconc parameter list is null.

Correct your program so the value is not null.

#### ICFRRCAREATOOSMALL

A program error occurred because the tpf\_cfconc answer area is too small as indicated by the length specified in the ANSLEN field of the tpf\_cfconc parameter list.

Correct your program and ensure that the length specified in the ANSLEN field accurately reflects the size of the tpf\_cfconc answer area. In addition, ensure that the length of the tpf\_cfconc answer area is large enough to contain the data returned.

#### ICFRRCBADAREA

A program error occurred because the pointer to the tpf\_cfconc answer area is null.

Correct your program so the value is not null.

#### ICFRRCNOLISTHDRS

A program error occurred because the number of list headers specified in the LISTHEADERS field of the tpf\_cfconc parameter list is zero.

Correct your program to use a value greater than zero.

#### ICFRRCCONNAME

A program error occurred because the connection name specified in the tpf\_cfconc parameter list matches the connection name of another active connection to the same CF structure. The connection name is used to identify your connection to a CF structure; therefore, the connection name must be unique for each connection to a CF structure.

Correct your program to use a connection name that is not already connected to the CF structure or allow the TPF system to generate a unique name for you.

**ICFRRCSTRTYPE**

A program error occurred because the CF structure type specified in the tpf\_cfconc parameter list does not match the type of the CF structure allocated previously. When you connect to an allocated CF structure, you cannot change the structure type attribute. The request fails.

Correct your program to use a CF structure type that matches the type specified for the original CF structure.

**ICFRRCCONNAMEERR**

A program error occurred because the connection name specified in the tpf\_cfconc parameter list is not valid.

Correct your program so that the connection name meets the following requirements:

- The name must begin with an uppercase alphabetic character.
- The name must be 16 characters long and padded on the right with blanks if necessary.
- The name can contain numeric characters, uppercase alphabetic characters, and underscores (\_).

**ICFRRCCFNAMEERR**

A program error occurred because the CF name specified in the tpf\_cfconc parameter list is not valid.

Correct your program so that the CF name meets the following requirements and then try the request again:

- The name must be a 5- to 8-character alphanumeric name.
- The first character must be an alphabetic character.

**ICFRRCSTRNAMEERR**

A program error occurred because the structure name specified in the tpf\_cfconc parameter list is not valid.

Correct your program so that the structure name meets the following requirements:

- The name must begin with an uppercase alphabetic character.  
TPF-defined structure names begin with I or TPF.
- The name must be 16 characters long, padded on the right with blanks if necessary.
- The name can contain numeric characters and uppercase alphabetic characters, and underscores (\_).

**ICFRRCINVALIDCACHEPARM**

A program error occurred because a value that is only valid for a CF list structure was specified for the CF cache structure.

Verify that the values specified are valid for a CF cache structure.

**ICFRRCINVALIDVECTORLEN**

A program error occurred because the vector length specified in the tpf\_cfconc parameter list for a CF cache structure was equal to zero. The request fails.

Correct your program by specifying a value of nonzero for the vector length for a CF cache structure.

**ICFRRCENTRYRATIO**

A program error occurred because the values specified in the ELEMENTRATIO and ENTRYRATIO fields of the tpf\_cfconc parameter list

are not valid. When the value specified in the ELEMENTRATIO field is greater than zero, the value specified in the ENTRYRATIO field must also be greater than zero.

Correct your program so that the value specified in the ENTRYRATIO field is greater than zero.

**ICFRRCMAXELEMNUM**

A program error occurred because the values specified in the ELEMENTRATIO and MAXELEMNUM fields of the tpf\_cfconc parameter list are not valid. When allocating a CF list structure, if the value specified in the ELEMENTRATIO field is not zero, the value specified in the MAXELEMNUM field must be greater than or equal to the ELEMENTRATIO value divided by the value you specified in the ENTRYRATIO field.

Correct your program so that the value specified for the MAXELEMNUM field meets these requirements.

**ICFRRCELEMINCRNUMELEMCHAR**

A program error occurred because nonzero values were specified for both the ICFCOELEMCHAR and ICFCOELEMINCRNUM fields of the tpf\_cfconc parameter list.

Correct your program to pass only one of these parameters and try the request again.

**ICFRRCELEMINCRNUM**

A program error occurred because the value specified in the ELEMINCRNUM field of the tpf\_cfconc parameter list is not valid.

Correct your program to specify a value greater than zero and a power of 2.

**ICFRRCMAXELEMNUMELEMCHAR**

A program error occurred because the values specified in the MAXELEMNUM field and either the ELEMCHAR or ELEMINCRNUM fields of the tpf\_cfconc parameter list will result in a list entry having a maximum data entry size greater than the 64-KB limit..

Correct your program to change the values of the MAXELEMNUM and ELEMCHAR fields, as appropriate, to meet the 64-KB limit.

**ICFRRCCONAUTHERR**

An attempt was made to connect to a TPF locking structure, but another processor has already connected to the structure with the processor ID of this processor.

Do the following:

1. Ensure the correct processor ID was specified for this processor during an initial program load (IPL).
2. Enter **ZPSMS PR FORCE DEACT** *processor ID* to ensure that any processor that was IPLed previously with the processor ID of this processor has been deactivated.

**ICFRRCNOMORECONNS**

An environmental error occurred because the CF structure already has the maximum number of allowed connections.

Try your request again at a later time.

**ICFRRCNOMORESTRUCTS**

An environmental error occurred because the CF cannot allocate a spare CF structure identifier (ID).

Try your request again at a later time.

#### **ICFRRINVSTRSIZE**

An environmental error occurred because the size specified for the CF structure was not valid.

Correct your program and try the request again.

#### **ICFRRCTRFAILURE**

An environmental error occurred because of a CF structure failure.

#### **ICFRRCREALLOCERROR**

An environmental error occurred because the second allocation to change the CF structure name from TPFALLOC to the requested name was not successful.

Disconnect from the CF structure.

#### **ICFRRCCFLEVEL**

An environmental error occurred because the requester specified a CF level that is greater than the level of the CF.

#### **ICFRRCAUTHLOCKERROR**

The connect request was rejected because an error prevented the CF lock from being obtained. This error occurs when the CF fails to respond to CF commands or the CF is damaged.

#### **ICFRRCCFNOTADDED**

A program error occurred because the CF was not added to the processor configuration.

Do the following:

1. Enter the ZMCFT ADD command to add the specified CF to the processor configuration.
2. Issue your request again.

See *TPF Operations* for more information about the ZMCFT ADD command.

#### **ICFRRCCFSBFILEERROR**

The connect request was not successful because the TPF system could not create the coupling facility structure block (CFSB) or update it on file. This error may be accompanied by additional error messages that provide more information about the error.

Correct the error and try to connect to the CF list structure again.

#### **ICFRRCCOMPERROR**

A component error occurred because of a TPF system processing failure.

Try the request again at a later time.

## **Programming Considerations**

- If the specified CF list or cache structure is not allocated when the tpf\_cfconc function is called, the CF list or cache structure is allocated and the connection is established.
- If the specified CF list or cache structure is already allocated when the tpf\_cfconc function is called, the TPF system connects to the existing CF structure.
- The CF structure may not be allocated with all the requested attributes. All connectors, whether the first or subsequent, are informed of the structure

## tpf\_cfconc

attributes through the tpf\_cfconc answer area. The connector must ensure the structure attributes are acceptable. If the structure attributes are not acceptable, you can disconnect from the CF structure.

- Applications should use the newCache function with the processor shared (Cache\_ProcS) parameter specified to create a CF cache structure rather than using the tpf\_cfconc function.

## Examples

The following example shows a connection to a CF list structure on a CF. In this example:

- The CF name is in the cf\_name variable.
- icfostrdisp is set to ICFCOKEEP, which means the structure disposition is KEEP.
- icfcotype is set to ICFCOLIST, which means this structure uses a CF list structure.
- Setting icfcostrctl means:
  - The list limits are tracked by ENTRY.
  - The list entries have an adjunct area.
  - Entries in the list will be located by name and not by key.
- The CF list structure name is in the structure\_name variable.
- The number of lists in this CF list structure is set by setting icfcolistheaders to list\_headers.
- Setting icfcovectorlen to 32 means lists will be monitored in the CF so a vector will be allocated.

```
#include <c$cfco.h>
#include <c$fapi.h>

struct icfco* parm_ptr = NULL;
struct icfcaa* answer_ptr = NULL;

/*****
/* Get storage for the connect parameter area and
/* the connect answer area. Initialize the pointers */
*****/
crusa( 2, D4, D5 );
parm_ptr = getcc( D4, GETCC_TYPE+GETCC_FILL, L1, 0x00 );
answer_ptr = getcc( D5, GETCC_TYPE+GETCC_FILL, L1, 0x00 );

/*****
/* Now perform the setup and call of tpf_cfconc()
*****/
parm_ptr->icfcoansarea = answer_ptr;
parm_ptr->icfcoanslen = _SBSZE;
parm_ptr->icfcocflvel = CFL_CFLEVEL;

memcpy( &parm_ptr->icfcocfname,
        cf_name,
        sizeof( parm_ptr->icfcocfname ) );

parm_ptr->icfcocompleteexit = complete_exit;

memcpy( &parm_ptr->icfcoconname,
        "GENERATED_NAME",
        sizeof( parm_ptr->icfcoconname ) );

parm_ptr->icfcostrdisp = ICFCOKEEP;
parm_ptr->icfcotype = ICFCOLIST;
```

```

parm_ptr->icfcostrctl =      (unsigned char)(ICFCOENTRY+
                                         ICFCOAJUNCT+ICFCONAMESUP+ICFCONOKEYSUP);

memcpy( &parm_ptr->icfcostrname,
        structure_name,
        sizeof( parm_ptr->icfcostrname ) );

parm_ptr->icfcostrsize = structure_size;
parm_ptr->icfcoelemchar =      0;
parm_ptr->icfcoelemincnum =    1;
parm_ptr->icfcoelementratio =  1;
parm_ptr->icfcoentryratio =    1;
parm_ptr->icfcolistheaders =   list_headers;
parm_ptr->icfcolisttranexit =  list_transition_exit;
parm_ptr->icfcolockentries =   0;
parm_ptr->icfcomaxelemnum =    1;
parm_ptr->icfcovectorlen =     32;

if( (rc = tpf_cfconc( parm_ptr ) != 0 )
{
    /* Handle the error or warning */

} /* End of error while connecting */

```

## Related Information

- “newCache—Create a New Logical Record Cache” on page 376
- “tpf\_cfdisc—Disconnect from a Coupling Facility List or Cache Structure” on page 570.

## tpf\_cfdisc–Disconnect from a Coupling Facility List or Cache Structure

This function disconnects you from a coupling facility (CF) list or cache structure when you no longer require access to it. A *CF list* or *cache structure* is a named piece of storage on a CF.

### Format

```
#include <c$fapi.h>
int tpf_cfdisc(struct icfdi *parms);
```

#### parms

A pointer to the tpf\_cfdisc parameter list.

### Normal Return

#### ICF\_SUCCESS

Disconnecting from a CF structure on a CF is successful. The connect token is no longer valid.

### Error Return

One of the following:

#### ICFRRCBADPARMLIST

A program error occurred because a pointer to the tpf\_cfdisc parameter list is null.

Correct your program so the value is not null.

#### ICFRRCBADCONTOKEN

A program error occurred because the connect token specified in the tpf\_cfdisc parameter list is not valid.

Correct your program to use the original connect token received in the tpf\_cfconc answer area after the connection request was issued.

#### ICFRRCCONNINUSE

A program error occurred because the requesting processor tried to disconnect from a CF structure that was still in use.

Let all outstanding requests to the CF structure end and issue the request again.

#### ICFRRCRESTARTINCOMPLETE

A program error occurred because CF restart has not yet completed. Be sure to wait until CF restart is complete.

#### ICFRRCAUTHLOCKERROR

The disconnect request was rejected because an error prevented the CF lock from being obtained. This error occurs when the CF fails to respond to CF commands or the CF is damaged.

### Programming Considerations

Applications should use the deleteCache function to discontinue use of a CF cache structure by a processor rather than using the tpf\_cfdisc function.

### Examples

The following example shows a normal tpf\_cfdisc call. The icfdicontoken parameter that is sent is the connect token of the CF list structure to which you were connected.



```

#include <c$cfdi.h>
#include <c$fapi.h>

struct icfdi* parm_ptr = NULL;
int rc = 0;

crusa( 1, D4 );
parm_ptr = getcc( D4, GETCC_TYPE+GETCC_FILL, L1, 0x00 )
memcpy( &parm_ptr->icfdicontoken,
        &cf_ptr->icflt_sst1.core_str.icflt_ssctk,
        ICF_CONTOKLEN );
rc = tpf_cfdisc( parm_ptr );
if( rc != ICFRRCOK )
{
    /* ERROR, WARNING OR BAD PARM ON DISC CALL */
}

```

## Related Information

- “deleteCache—Delete a Logical Record Cache” on page 81
- “tpf\_cfconc—Connect to a Coupling Facility List or Cache Structure” on page 564.

## tpf\_cresc—Create Synchronous Child ECBs

This function creates an ECB for immediate processing by the requested program. The ECB is created in the same subsystem and subsystem user as the parent ECB.

When exiting, the child ECB will notify the parent by posting an event on which the parent is waiting for completion. Posting the event gives the child ECB the ability to pass the return code value back to the parent ECB.

The tpf\_cresc function can be used to allow the parent ECB to create and then wait for multiple child ECBs to be completed by coding it multiple times, setting tci.wait = TPF\_CRESC\_WAIT\_YES on in the last entry. This will result in the creation of the child ECBs and will place the parent ECB in a wait state until all child ECBs have been completed or until they time out.

## Format

```
#include <tpfapi.h>
struct tpf_evobk_list_data *tpf_cresc(const struct tpf_cresc_input *tci);
```

The tpf\_cresc\_input structure consists of the following members:

### program

A pointer to the name of the program that will be activated with the created entry control block (ECB).

### istream

The I-stream number on which the child ECB will be dispatched. The following symbols are defined in the tpfapi.h file for use in the **istream** member:

TPF_CRESC_IS_MAIN	Create the child ECB on the main I-stream.
TPF_CRESC_IS_MPIF	Create the child ECB on the MPIF I-stream.
TPF_CRESC_IS_SAME	Create the child ECB on the same I-stream as the parent.
TPF_CRESC_IS_BALANCE	Create the child ECB on the least busy I-stream.

### timeout

A value, from 0 to 32 768, which determines how many seconds the parent ECB waits for the child ECBs to end. If a 0 value is given, the parent will wait indefinitely until the child ECBs exit. This member is ignored unless the wait member is set to TPF\_CRESC\_WAIT\_YES. This value will be in effect in 1052 state.

### wait

Determines if the child ECBs should be created and if the parent ECB should wait the length of the value specified for them to be completed. The following symbols are defined in the tpfapi.h header file for use in the **wait** member:

TPF_CRESC_WAIT_NO	Do not create the child ECBs, just initialize the creation information of the child ECB.
TPF_CRESC_WAIT_YES	Create and dispatch the child ECBs and wait for them to be completed.

### data

A pointer to the data to be passed to the child ECB. If no data will be passed to the child ECB, set a NULL pointer.

### data\_length

The number of bytes of data to be passed to the child ECB. If no data will be passed to the child ECB, set this field to zero.

## Normal Return

A pointer to the event block with list item data (see the `tpf_ev0bk_list_data` structure) is returned following a `tpf_cresc` call that specifies `tc_i.wait = TPF_CRESC_WAIT_YES`. The `tpf_ev0bk_list_item` structure will point to the start of the list where all child ECB return codes will be stored. The application must set up the child ECB return codes that are used.

A NULL pointer is returned following a `tpf_cresc` call that specifies `tc_i.wait = TPF_CRESC_WAIT_NO`.

## Error Return

If any of the following conditions are met, a SNAPC error is issued and the ECB exits:

- The value specified for the `istream` parameter is incorrect.
- The `MALOC` macro that is issued by the `CRESC` macro fails because there is not enough heap storage.

## Programming Considerations

- The application program can request that a maximum of 50 child ECBs can be created using this function. If more than 50 child ECBs are requested, a system error will occur.
- For more information on `TPF_EVENT_LIST_ITEM` and `TPF_EVENT_DATA_SIZE`, see the `tpfapi.h` header file.
- If you use this macro to create an ECB that will enter a dynamic load module (DLM) with an entry point defined by the `main` function, the TPF system assumes that any core block attached to data level 0 (D0) contains a command string that will be parsed into `argc` and `argv` parameters for the `main` function. See *TPF Application Programming* for more information about the `main` function.

**Note:** Limit the use of this function to prevent storage depletion.

## Examples

The following example creates two child ECBs and places the parent ECB in a wait state until both child ECBs have completed processing or until 100 seconds have elapsed.

When the parent ECB has been reactivated, it interrogates the list item data that has been appended to the event block to determine the value returned by the child ECBs.

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

int QXQZ(void)
{
    /******
    /* Variable definitions
    /******

    struct tpf_cresc_input tci;          /* tpf_cresc I/P parameter */
                                       /* structure */

    struct tpf_ev0bk_list_data *ebptra; /* Pointer to Event block */
                                       /* with list item data */
```

## tpf\_cresc

```
tpf_ev0bk_list_item *liptr;          /* Pointer to list item data */

int index;                            /* Index for accessing list */
                                     /* item data */

/*****
/* Initialize parameters for CRESC WAIT=NO call.
*****/

char data[11] = "Sample data";        /* Data to be passed */

tci.program = "QPM1";                 /* Pgm for child to enter */
tci.istream = 1;                      /* I/S for child to run on */
tci.wait = TPF_CRESC_WAIT_NO;         /* Don't create ECB yet */
tci.data_length = 11;                 /* Pass 11 bytes of data */
tci.data = data;                      /* Pass data assigned to */
                                     /* variable data */

/*****
/* Issue CRESC WAIT=NO call.
*****/

ebpтр = tpf_cresc(&tci);               /* Issue WAIT=NO call which */
                                     /* returns a NULL pointer */

/*****
/* Initialize parameters for CRESC WAIT=YES call.
*****/

tci.program = "QPM1";                 /* Pgm for child to enter */
tci.istream = TPF_CRESC_IS_MAIN;      /* I/S for child */
tci.timeout = 300;                    /* Timeout value for parent */
                                     /* Parent will wait forever */
tci.wait = TPF_CRESC_WAIT_YES;        /* Create all children */
tci.data_length = 0;                  /* No data being passed */
tci.data = NULL;                      /* No data being passed */

/*****
/* Issue CRESC WAIT=YES call. All child ECBs requested to date will */
/* now be created and dispatched.
*****/

ebpтр = tpf_cresc(&tci);               /* Issue WAIT=YES call which */
                                     /* will return a pointer to */
                                     /* the Event block created */

/*****
/* Invoke the TPF_EVENT_LIST_ITEM macro to access the list items */
/* appended to the Event block created for this event.
/*
/* The list item, one for each child created, will contain */
/* information on whether the item has been posted (child has */
/* exited), the item has been posted with an error (child exited */
/* due to a system error), and the return code set by the child ECB.*/
/*
/*
/* exit(4);                          This is how the child */
                                     /* ECB exits and returns a */
                                     /* value to the parent ECB. */

/* If the item has been posted, check to see if an error has been */
/* posted. If so, issue the message 'Child exited with error'. If */
/* no error has been posted, check the return code set by the child.*/
/* If the return code = 4, issue the message 'Test case successful'.*/
/* If the return code != 4, issue the message 'Test case */
/* unsuccessful.' If the item has not been posted, issue the */
```

```

/* message 'The item is not posted because the timeout value was */
/* exceeded.' */
/*****

for (index = 0; index < ebp->evnbklc; index = index+1)
{
    liptr = TPF_EVENT_LIST_ITEM(ebptr,index); /* Call macro to return */
                                              /* address of list item data */

    if ((liptr->evnbklif & TPF_EVNBK_POST) != 0) /* Item posted? */
        if ((liptr->evnbklif & TPF_EVNBK_ECDE) != 0) /* Yes, posted */
                                                    /* with error? */
        {
            printf("Child exited with error"); /* Yes, issue msg. */
        }
        else
        {
            if (liptr->evnbkvlis == 4) /* No, check RC = 4 */
            {
                printf("Test case successful"); /* Yes, successful */
            }
                                                    /* No, unsuccessful */
            else printf("Test case unsuccessful.");
        }
    }
    else
        printf("The item is not posted because the timeout value was exceeded.");
}

exit(0); /* Exit */
return 0; /* Avoid compiler warning */
}

```

## Related Information

- “credc, \_\_CREDC—Create a Deferred Entry” on page 57
- “creec, \_\_CREEC—Create a New ECB with an Attached Block” on page 59
- “cremc, \_\_CREMC—Create an Immediate Entry” on page 62
- “cretc, \_\_CRETc—Create a Time-Initiated Entry” on page 64
- “crexc, \_\_CREXC—Create a Low-Priority Deferred Entry” on page 69.

## tpf\_decb\_create—Create a Data Event Control Block

This function provides interfaces to create a data event control block (DECB).

### Format

```
#include <c$decb.h>
TPF_DECB *tpf_decb_create(char *name, DECBC_RC *rc);
```

#### name

A pointer to a 16-byte user-specified DECB name. The name parameter is optional. If NULL is coded, a name is not assigned to the DECB.

**rc** A pointer to the return code. **rc** is an optional parameter and, if NULL is coded, the return code will not be set.

### Normal Return

A pointer to a DECB and the return code is set to DECBC\_OK.

### Error Return

A NULL pointer and **rc** contains DECBC\_DUPNAME if the name parameter specified contains a DECB name that already exists for this entry control block (ECB).

## Programming Considerations

- The DECB is an alternative to standard ECB data level information, which is used to specify information about I/O request core block reference word (CBRW) and file address reference word (FARW) fields. The DECB fields specify the same CBRW and FARW information without requiring the use of an ECB data level. All the same requirements and conditions that apply to the CBRW and FARW in the ECB also pertain to the same field information in the DECB.
- The FARW in the DECB provides storage for an 8-byte file address.
- Functions that support the use of a DECB (such as `file_record_ext`, `find_record_ext`, and so on) will only accept a DECB address as a valid reference to a DECB. If an application does not maintain the address of a particular DECB and instead maintains the name of the DECB, the caller will first have to issue the `tpf_decb_locate` function to obtain the address of the desired DECB. The resulting DECB address may then be passed on the subsequent function call.
- Applications that use DECBs must be compiled with the C++ compiler.
- The letters l, i, TPF, TPF\_, tpf, and tpf\_ are reserved for future use by IBM for DECB name spaces.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example creates a DECB with a given name.

```
#include <c$decb.h>
:
DECBC_RC rc;
TPF_DECB *decb;
char decb_name[16] = "APPLWXY";
:
if ( (decb = tpf_decb_create(decb_name, &rc)) != NULL );
```

```
{
    /* DECB is successfully created */
} else
{
    /* failed to create DECB, check rc for the reason */
}
```

## Related Information

- “tpf\_decb\_locate—Locate a Data Event Control Block” on page 578
- “tpf\_decb\_release—Release a Data Event Control Block” on page 580
- “tpf\_decb\_swapblk—Swap a Storage Block with a Data Event Control Block” on page 582
- “tpf\_decb\_validate—Validate a Data Event Control Block” on page 584.

See *TPF Application Programming* for more information about DECBs.

## tpf\_decb\_locate—Locate a Data Event Control Block

This function provides interfaces to locate a data event control block (DECB).

### Format

```
#include <c$decb.h>
TPF_DECB *tpf_decb_locate(char *name, DECBC_RC *rc);
```

or

```
#include <c$decb.h>
TPF_DECB *tpf_decb_locate(TPF_DECB *decb, DECBC_CHAIN chain, DECBC_RC *rc);
```

#### name

A pointer to a 16-byte user-specified DECB name. The DECB name was previously specified using the `tpf_decb_create` function.

#### decb

A pointer to the current DECB. If NULL is specified, the first DECB, as specified by the chain parameter, will be returned.

#### chain

Indicates whether an active DECB or any DECB will be returned and is one of the following:

##### DECBC\_CHAIN\_INUSE

Indicates the next active DECB after the current DECB will be returned.

##### DECBC\_CHAIN\_ANY

Indicates the next DECB, active or inactive, after the current DECB will be returned.

**rc** A pointer to the return code. **rc** is an optional parameter and, if NULL is coded, the return code will not be set.

### Normal Return

A pointer to a DECB and the return code is set to DECBC\_OK.

### Error Return

A NULL pointer and **rc** contains DECBC\_NOTFOUND if the name parameter specified contains a DECB name that is not valid for this entry control block (ECB).

### Programming Considerations

- The DECB is an alternative to standard ECB data level information, which is used to specify information about I/O request core block reference word (CBRW) and file address reference word (FARW) fields. The DECB fields specify the same CBRW and FARW information without requiring the use of an ECB data level. All the same requirements and conditions that apply to the CBRW and FARW in the ECB also pertain to the same field information in the DECB.
- The FARW in the DECB provides storage for an 8-byte file address.
- Functions that support the use of a DECB (such as `file_record_ext`, `find_record_ext` and so on) will only accept a DECB address as a valid reference to a DECB. If an application does not maintain the address of a particular DECB and instead maintains the name of the DECB, the caller will first have to issue the `tpf_decb_locate` function to obtain the address of the DECB. The resulting DECB address may then be passed on the subsequent function call.
- Applications that use DECBs must be compiled with the C++ compiler.



- The letters l, i, TPF, TPF\_, tpf, and tpf\_ are reserved for future use by IBM for DECB name spaces.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example locates a DECB with a given name.

```
#include <c$decb.h>
...
DECBC_RC   rc;
TPF_DECB   *decb;
char        decb_name[16] = "APPLWXY";
...
if ( (decb = tpf_decb_locate(decb_name, &rc)) != NULL );
{
    /* DECB is successfully located */
}
else
{
    /* failed to locate DECB, check rc for the reason */
}
```

## Related Information

- “tpf\_decb\_create—Create a Data Event Control Block” on page 576
- “tpf\_decb\_release—Release a Data Event Control Block” on page 580
- “tpf\_decb\_swapblk—Swap a Storage Block with a Data Event Control Block” on page 582
- “tpf\_decb\_validate—Validate a Data Event Control Block” on page 584.

See *TPF Application Programming* for more information about DECBs.

## tpf\_decb\_release—Release a Data Event Control Block

This function provides interfaces to release a data event control block (DECB).

### Format

```
#include <c$decb.h>
void tpf_decb_release(TPF_DECB *decb);
```

or

```
#include <c$decb.h>
void tpf_decb_release (char *name);
```

#### decb

A pointer to the DECB to be released.

#### name

A pointer to a 16-byte user-specified DECB name. The DECB name was previously specified on a tpf\_decb\_create call.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The DECB is an alternative to standard ECB data level information, which is used to specify information about I/O request core block reference word (CBRW) and file address reference word (FARW) fields. The DECB fields specify the same CBRW and FARW information without requiring the use of an ECB data level. All the same requirements and conditions that apply to the CBRW and FARW in the ECB also pertain to the same field information in the DECB.
- The FARW in the DECB provides storage for an 8-byte file address.
- The service routine will take a SERRC (which is a standard memory dump of main storage) and exit the ECB for any of the following reasons:
  - The DECB to be released does not currently exist
  - The DECB is holding a block of storage
  - The DECB has I/O pending
  - The DECB has detached core blocks.
- Applications that use DECBs must be compiled with the C++ compiler.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example locates a DECB and releases it.

```
#include <c$decb.h>
...
DECB_RC rc;
TPF_DECB *decb;
char decb_name[16] = "APPLWXY";
...
if ( (decb = tpf_decb_locate(decb_name, &rc)) != NULL );
```

```
{
    tpf_decb_release(decb); /* release the DECB that was located */
} else
{
    /* failed to locate DECB, check rc for the reason */
}
```

## Related Information

- “tpf\_decb\_create—Create a Data Event Control Block” on page 576
- “tpf\_decb\_locate—Locate a Data Event Control Block” on page 578
- “tpf\_decb\_swapblk—Swap a Storage Block with a Data Event Control Block” on page 582
- “tpf\_decb\_validate—Validate a Data Event Control Block” on page 584.

See *TPF Application Programming* for more information about DECBs.

## tpf\_decb\_swapblk—Swap a Storage Block with a Data Event Control Block

This function swaps the storage block that is held in a data event control block (DECB) with an entry control block (ECB) data level.

### Format

```
#include <c$decb.h>
void tpf_decb_swapblk(TPF_DECB *decb, enum t_lvl level);
```

#### decb

A pointer to the DECB.

#### level

One of 16 possible values representing a valid ECB data level from enumeration type `t_lvl`, expressed as *Dx*, where *x* represents the hexadecimal number of the level (0–F).

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- The DECB is an alternative to standard ECB data level information, which is used to specify information about I/O request core block reference word (CBRW) and file address reference word (FARW) fields. The DECB fields specify the same CBRW and FARW information without requiring the use of an ECB data level. All the same requirements and conditions that apply to the CBRW and FARW in the ECB also pertain to the same field information in the DECB.
- Applications that use DECBs must be compiled with the C++ compiler.
- Only the CBRW is swapped; the FARW in both the ECB data level and DECB remain unchanged.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

### Examples

The following example swaps the core block and associated information between ECB data level 2 (D2) and the indicated DECB.

```
#include <c$decb.h>
...
TPF_DECB *decb;
...
tpf_decb_swapblk(decb, D2); /* swap storage block on decb and D2 */
...
```

### Related Information

- “tpf\_decb\_create—Create a Data Event Control Block” on page 576
- “tpf\_decb\_locate—Locate a Data Event Control Block” on page 578
- “tpf\_decb\_release—Release a Data Event Control Block” on page 580
- “tpf\_decb\_validate—Validate a Data Event Control Block” on page 584.

See *TPF Application Programming* for more information about DECBs.

## tpf\_decb\_validate—Validate a Data Event Control Block

This function validates that a specified data event control block (DECB) is an active and valid DECB.

### Format

```
#include <c$decb.h>
DECBC_RC tpf_decb_validate(TPF_DECB *decb);
```

#### decb

A pointer to the DECB.

### Normal Return

DECBC\_OK.

### Error Return

DECBC\_NOTVALID.

## Programming Considerations

- The DECB is an alternative to standard ECB data level information, which is used to specify information about I/O request core block reference word (CBRW) and file address reference word (FARW) fields. The DECB fields specify the same CBRW and FARW information without requiring the use of an ECB data level. All the same requirements and conditions that apply to the CBRW and FARW in the ECB also pertain to the same field information in the DECB.
- Applications that use DECBs must be compiled with the C++ compiler.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example validates a DECB address.

```
#include <c$decb.h>
...
TPF_DECB *decb;
...
if ( tpf_decb_validate(decb) == DECBC_OK )
{
    /* DECB is valid */
} else
{
    /* DECB is not valid */
}
```

## Related Information

- “tpf\_decb\_create—Create a Data Event Control Block” on page 576
- “tpf\_decb\_locate—Locate a Data Event Control Block” on page 578
- “tpf\_decb\_release—Release a Data Event Control Block” on page 580
- “tpf\_decb\_swapblk—Swap a Storage Block with a Data Event Control Block” on page 582.

See *TPF Application Programming* for more information about DECBs.

## tpf\_dlckc – Modify Lock and Input/Output Interrupt Status

This function:

- Disables input/output (I/O) interrupts and locks a resource
- Unlocks a resource and enables I/O interrupts.

### Format

```
#include <sysapi.h>
tpf_dlckc(*lock_pointer, lock_function)
```

#### **lock\_pointer**

A pointer to the address of a doubleword lock field.

#### **lock\_function**

Specify one of the following:

##### **DLCKC\_LOCK**

The lockword specified by the `lock_pointer` parameter is locked and I/O interrupts are disabled on the I-stream calling this function.

##### **DLCKC\_UNLOCK**

The lockword specified by the `lock_pointer` parameter is unlocked and I/O interrupts are enabled on the I-stream calling this function.

### Normal Return

None.

### Error Return

None.

### Programming Considerations

This function can be used only by ECB-controlled programs.

### Examples

The following example shows a resource being locked with the I/O interrupts disabled and then unlocked with the I/O interrupts enabled.

```
#include <sysapi.h>
struct mflsec* mfs1_ptr = NULL;
mfs1_ptr = cinfc(CINFC_WRITE,CINFC_CMMFS1);

tpf_dlckc( mfs1_ptr->mfliolkp, DLCKC_LOCK );

:
tpf_dlckc( mfs1_ptr->mfliolkp, DLCKC_UNLOCK );
```

### Related Information

None.

## tpf\_esfac—Obtain Extended Symbolic File Address Information

This function obtains information about the characteristics of a specific symbolic file address.

### Format

```
#include <sysapi.h>
int tpf_esfac(unsigned int file_address, tpf_sonfmt *son_info);
```

or

```
#include <sysapi.h>
int tpf_esfac(TPF_FA8 *fa8, tpf_sonfmt *son_info);
```

#### **file\_address**

A file address for which the information is to be obtained.

#### **fa8**

A pointer to an 8-byte file address for which the information is to be obtained.

#### **son\_info**

A pointer to an empty record where the information is to be kept upon return.

### Normal Return

Integer value of 0 indicating successful completion.

### Error Return

A nonzero value is returned if the file address information could not be obtained.

### Programming Considerations

- This function requires that the application program allocation specify authorization to issue a restricted macro.
- Applications that call this function using 8-byte file addresses instead of entry control block (ECB) data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

### Examples

The following example returns the file characteristics of the file address reference format (FARF) address in ECB data level 2 and for the specified 8-byte file address.

```
#include <sysapi.h>
:
:
int retval;
tpf_sonfmt son_info;
TPF_FA8 fa8;
:
:
retval = tpf_esfac((const unsigned int)ecbptr()->ebcfa2, &son_info);
:
:
retval = tpf_esfac(&fa8, &son_info);
```

### Related Information

“sonic—Obtain Symbolic File Address Information” on page 511.



## tpf\_fac8c—Low-Level File Address Compute Functions

This function returns an 8-byte file address based on the input record type and an 8-byte ordinal number.

### Format

```
#include <tpfio.h>
int tpf_fac8c(TPF_FAC8 *parm);
```

#### parm

A pointer to a parameter block described by the TPF\_FAC8 structure. Fields in the input area of this block that are to be provided by the caller must be initialized before the function is called. Output area fields will be filled in by the function service routine.

### Normal Return

TPF\_FAC8\_NRM. The **ifacret** field in the TPF\_FAC8 structure also contains TPF\_FAC8\_NRM, and the **ifacmax** field in the TPF\_FAC8 structure contains the maximum ordinal number for the requested record type and the file address has been placed in the field **ifacadr** in the TPF\_FAC8 structure.

### Error Return

One of the following:

TPF_FAC8_NIU	Record type is not in use.
TPF_FAC8_RTH	Record type does not exist or exceeds the FACE table limit.
TPF_FAC8_ROR	Record ordinal is outside the allowable range.
TPF_FAC8_NSP	No split chain for the record.
TPF_FAC8_POR	Input parameter is outside the allowable range.

The **ifacret** field in the TPF\_FAC8 structure also contains the return code.

### Programming Considerations

- Applications that use 8-byte file addresses must be compiled with the C++ compiler.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

### Examples

The following example generates a system ordinal number (SON) address for #RECTYPE record number 198 and stores it in the output area of the TPF\_FAC8 block.

```
#include <tpfio.h>
:
:
TPF_FAC8 fa;
MCHR_STRUCT mchr;

/* call tpf_fac8c to calculate FARF address */
fa.ifacord = 198;
memcpy(fa.ifacrec, "#RECTYPE", sizeof(fa.ifacrec));
fa.ifactyp=IFAC8FCS;
tpf_fac8c(&fa);
```

## tpf\_fac8c

```
if (fa.ifacret != TPF_FAC8_NRM)
{
    /* error return */
}
else
{
    /* normal return */
}
```

## Related Information

“face\_fac8c—File Address Generation” on page 120.

## tpf\_faczc–File Address Calculation

This function provides the interface to the file address compute program (FACE) address generation routines and allows access to records that are unique to a subsystem, subsystem user (SSU), processor, or I-stream.

### Format

```
#include <sysapi.h>
int tpf_faczc(IDSFCZ *idsfcz,
              FCZ_YES_NO default_SS,
              FCZ_YES_NO default_SSU,
              FCZ_YES_NO default_proc,
              FCZ_YES_NO default_IStream,
              FCZ_REQUEST request);
```

#### idsfcz

A pointer to the IDSFCZ parameter block that holds the input and output of the FACZC system service.

#### default\_SS

Specifies if a default subsystem index is to be used as input to the service, where:

- FCZ\_YES - The default subsystem index in the entry control block (ECB) will be used as input.
- FCZ\_NO - The user-provided subsystem index in the IDSFCZ parameter block will be used as input.

#### default\_SSU

Specifies if the default SSU index is to be used as input to the service, where:

- FCZ\_YES - The default SSU index in the ECB will be used as input.
- FCZ\_NO - The user-provided SSU index in the IDSFCZ parameter block will be used as input.

#### default\_proc

Specifies if a default processor ordinal number in the ECB will be used as input to the service, where:

- FCZ\_YES - The default processor ordinal number in the ECB will be used as input.
- FCZ\_NO - The user-provided processor ordinal number in the IDSFCZ parameter block will be used as input.

#### default\_IStream

Specifies if the default I-stream number in the ECB will be used as input to the service, where:

- FCZ\_YES - The default I-stream number in the ECB will be used as input.
- FCZ\_NO - The user-provided processor I-stream number in the IDSFCZ parameter block will be used as input.

#### request

Specifies the type of request, where:

- FCZ\_USER - The user has to specify the request type and set up the correct record ID in the IDSFCZ parameter block as input.
- FCZ\_FACE - Indicates a FACE-type call. The record type number must be placed in the IDSFCZ parameter block input area.
- FCZ\_FACS - Indicates a FACE-type call. The 8-character record name (padded with blanks) must be placed in the IDSFCZ parameter block input area.

## tpf\_faczc

- FCZ\_FACE8 - indicates a FAC8C-type call. The 8-byte ordinal number and record type number must be placed in the IDSFCZ parameter block input area.
- FCZ\_FACS8 - indicates a FAC8C-type call. The 8-byte ordinal number and 8-byte character record type name (padded with blanks) must be placed in the IDSFCZ parameter block input area.

## Normal Return

FCZRC\_OK.

## Error Return

### FCZRC\_TYPE\_NOT\_IN\_USE

The requested record type is not in use.

### FCZRC\_OUT\_OF\_RANGE

The requested record type does not exist or exceeds the limit.

### FCZRC\_ORD\_OUT\_OF\_RANGE

The record ordinal number is out of the allowable range.

### FCZRC\_NO\_SPLIT\_CHAIN

The record has no split chain.

### FCZRC\_PARM\_OUT\_OF\_RANGE

The input parameter is outside the allowable range.

## Programming Considerations

- This function requires that the application program allocation specify authorization to issue a restricted macro.
- A symbolic record type (FACS or FAC8C interface) or record type number (FACE interface) and an ordinal within that record type must be provided. The type of call (FACS, which is the default, FACE, FACE8, or FAC8C) is specified by using the **request** parameter. Optionally, a subsystem, subsystem user, processor, or I-stream (or any combination of these) can be provided.

## Examples

The following example generates a file address for "#PROG1" record number 235.

```
#include <sysapi.h>
...
IDSFCZ f={0};
f.fcz_ordinal=235;
memcpy(f.fcz_rec_type.name, "#PROG1 ",8);
f.fcz_request_type =FCZ_REQ_FACS;
tpf_faczc(&f, FCZ_YES, FCZ_YES, FCZ_YES, FCZ_YES, FCZ_USER);
```

## Related Information

- "FACE, FACS—Low-Level File Address Compute Functions" on page 117
- "filec—File a Record: Basic" on page 152
- "filec\_ext—File a Record with Extended Options: Basic" on page 154
- "file\_record—File a Record: Higher Level" on page 158
- "file\_record\_ext—File a Record with Extended Options: Higher Level" on page 161
- "filnc—File a Record with No Release" on page 165
- "filnc\_ext—File a Record with No Release and Extended Options" on page 167

- “filuc–File and Unhold a Record” on page 170
- “filuc\_ext–File and Unhold a Record with Extended Options” on page 172
- “findc–Find a Record” on page 174
- “findc\_ext–Find a Record with Extended Options” on page 176
- “find\_record–Find a Record” on page 178
- “find\_record\_ext–Find a Record with Extended Options” on page 181
- “finhc–Find and Hold a File Record” on page 185
- “finhc\_ext–Find and Hold a File Record with Extended Options” on page 187
- “finwc–Find a File Record and Wait” on page 189
- “finwc\_ext–Find a File Record and Wait with Extended Options” on page 191
- “fiwhc–Find and Hold a File Record and Wait” on page 193
- “fiwhc\_ext–Find and Hold a File Record and Wait with Extended Options” on page 195.

## tpf\_fa4x4c—Convert a File Address

This function does the following:

- Converts a 4-byte file address to an 8-byte file address in 4x4 format
- Converts an 8-byte file address in 4x4 format to a 4-byte file address.

**Note:** 4x4 format is an 8-byte file address with a high-order 4-byte indicator (that contains zeros) and a low-order 4 bytes that contains a 4-byte FARF3, FARF4, or FARF5 address.

## Format

```
#include <tpfio.h>
int      tpf_fa4x4c(TPF_FACONV action, unsigned long *fa4,
                  TPF_FA8 *fa8);
```

### action

One of the following action types:

#### FACONV\_4TO8

Converts a 4-byte file address to an 8-byte file address in 4x4 format.

#### FACONV\_8TO4

Converts an 8-byte file address in 4x4 format to a 4-byte file address.

### fa4

A pointer to a 4-byte file address. This parameter is the input if the action is FACONV\_4TO8. This parameter is the output if the action is FACONV\_8TO4.

### fa8

A pointer to an 8-byte file address in 4x4 format. This parameter is the input if the action is FACONV\_8TO4. This parameter is the output if the action is FACONV\_4TO8.

## Normal Return

A nonzero value.

## Error Return

An integer value of zero. This can occur if FACONV\_8TO4 is specified and the input 8-byte file address is not in 4x4 format, or if the specified action is not valid.

## Programming Considerations

None.

## Examples

The following example converts a 4-byte file address to an 8-byte file address and converts an 8-byte file address to a 4-byte file address.

```
#include <tpfio.h>
...
unsigned long  fa4;
TPF_FA8 fa8;
...
tpf_fa4x4c(FACONV_4TO8, &fa4, &fa8); /* convert 4-byte fa to 8-byte */
...
tpf_fa4x4c(FACONV_8TO4, &fa4, &fa8); /* convert 8-byte fa to 4-byte */
```

## Related Information

None.

## tpf\_fork—Create a Child Process

This function creates a child process on a specified I-stream. The caller is called the *parent process* and the created entry control block (ECB) is called the *child process*. The process identifier (ID) of the child process is returned by this function, while the process ID of the caller is used as the *parent process ID* of the child process. Using process IDs, the parent and child processes can use the `kill` function to send signals to each other. Additionally, when the child process exits, a SIGCHLD signal is sent to the parent process automatically.

### Format

```
#define TPF_FORK_EXTENDED
#include <sysapi.h>
pid_t tpf_fork (const struct tpf_fork_input *create_parameters,
               const char *argv[ ], const char *envp[ ]);
```

or

```
#include <sysapi.h>
pid_t tpf_fork(const struct tpf_fork_input *create_parameters);
```

#### create\_parameters

A pointer to a fully initialized `tpf_fork_input` structure. The following fields are defined in the `tpf_fork_input` structure:

##### program

A pointer to the name of the first program to be called by the created ECB. The program must be a dynamic load module (DLM) and its entry point must be a main function. It can be specified in the following ways:

- If **prog\_type** is **TPF\_FORK\_NAME**, **program** points to a 4-character TPF program name.
- If **prog\_type** is **TPF\_FORK\_FILE**, **program** points to the path name of a file system executable file that contains a reference to a TPF program. This path name can specify a relative path or an absolute path, which is indicated by a preceding slash. The file must contain one of the following:
  - `#!nnnn`, where `nnnn` is the 4-character name of a TPF program to be run
  - `#!pathname`, where `pathname` is the path to the file that contains `#!nnnn`.
- If **prog\_type** is **TPF\_FORK\_AND\_SEARCH**, **program** points to the path name of a file system executable file that contains a reference to a TPF program. This path name can specify a relative path or an absolute path that is indicated by a preceding slash. Otherwise, the directories specified by the PATH environment variable are searched for the file system executable file. The file must contain one of the following:
  - `#!nnnn`, where `nnnn` is the 4-character name of a TPF program to be run
  - `#!pathname`, where `pathname` is the path to the file that contains `#!nnnn`.
- If **prog\_type** is **TPF\_FORK\_FILE** or **TPF\_FORK\_AND\_SEARCH** and **program** points to the path name of a file system executable file that does not contain a reference to a TPF program in the form `#!nnnn`, the `tpf_fork` function calls segment CFIX, which is the default executable ZFILE parser, by using the path name of the caller and the remaining input arguments. This is similar to the following:



```

struct tpf_fork_input create_parameters;
create_parameters.program = "CFIX";
create_parameters.prog_type = TPF_FORK_NAME;
create_parameters.istream = TPF_FORK_IS_BALANCE;
create_parameters.ebw_data = NULL;
create_parameters.ebw_data_length = 0;
create_parameters.parm_data = NULL;
argv[0] = "sh";
argv[1] = "-c";
argv[2] = fully_expanded_pathname;
for ( i=3; argv[3-i] != NULL; ++i)
    argv[i] = argv[3-i];
argv[i] = NULL;
tpf_fork( &create_parameters, argv, NULL );

```

where `argv` is an array containing all the caller's arguments to `tpf_fork`, and `fully_expanded_pathname` is the path name of the script found by searching the directories in the current path.

### prog\_type

One of the following:

- **TPF\_FORK\_NAME**, if **program** specifies a 4-character TPF program name.
- **TPF\_FORK\_FILE**, if **program** specifies the path name of a file system executable file that refers to a TPF program. This path name can specify a relative path or an absolute path, which is indicated by a preceding slash.
- **TPF\_FORK\_AND\_SEARCH**, if **program** specifies the path name of a file system executable file that refers to a TPF program. This path name can specify a relative path or an absolute path, which is indicated by a preceding slash. Otherwise, the file system executable is searched in the directories specified by the `PATH` environment variable.

### istream

The I-stream where the asynchronous entry is created; specify either the I-stream number (1-based ordinal) or one of the following:

#### TPF\_FORK\_IS\_MAIN

Create the asynchronous entry on the main I-stream. Specify either this option or the I-stream number (1-based ordinal).

#### TPF\_FORK\_IS\_MPIF

Create the asynchronous entry on the Multi-Processor Interconnect Facility (MPIF) I-stream. Specify either this option or the I-stream number (1-based ordinal).

#### TPF\_FORK\_IS\_BALANCE

Create the asynchronous entry on the least busy I-stream. Specify either this option or the I-stream number (1-based ordinal).

### ebw\_data\_length

The number of bytes of data (0–104 bytes) to be passed to the child process.

### ebw\_data

A pointer to void containing the address of data to be passed to the child process beginning at `EBW000`.

### parm\_data

A pointer to additional data to be passed to **program**. The data is passed to the main function in **program** in the **argv** array. The total length of the

## tpf\_fork

**program** and **parm\_data** strings must be less than 4094 bytes. If **argv** is supplied, **parm\_data** must equal NULL. Otherwise, an error is returned.

### argv

is a pointer to an array of pointers to null-terminated character strings. There must be a NULL pointer after the last character string to mark the end of the array. These strings are used as arguments for the process being called. **argv[0]** should point to a string containing the name of a file (program name) associated with the process being started by **tpf\_fork**. If **parm\_data** is supplied, **argv** must equal NULL. Otherwise, an error is returned.

### envp

is a pointer to an array of pointers to null-terminated character strings. There must be a NULL pointer after the last character string to mark the end of the array. The strings of **envp** provide the environment variables for the new process.

## Normal Return

The process ID of the created child process is returned.

## Error Return

A value of -1 and **errno** is set to one of the following:

<b>EACCESS</b>	The program does not have permission to run the specified file system path name.
<b>EINVAL</b>	The <b>create_parameters</b> parameter is a NULL pointer, or one or more of the members of the referenced struct <b>tpf_fork_input</b> object are not valid: <ul style="list-style-type: none"><li>• The <b>program</b> member is a NULL pointer.</li><li>• The <b>ebw_data_length</b> member is less than zero or greater than 104.</li><li>• The <b>ebw_data_length</b> member is greater than zero and less than or equal to 104, and the <b>ebw_data</b> member is a NULL pointer.</li><li>• The <b>prog_type</b> member is not <b>TPF_FORK_NAME</b>, <b>TPF_FORK_FILE</b>, or <b>TPF_FORK_AND_SEARCH</b>.</li></ul>
<b>ELOOP</b>	A loop exists in the symbolic links. This error occurs if the number of symbolic links detected in the resolution is greater than <b>POSIX_SYMLINK_LOOP</b> (a value defined in the <b>limits.h</b> header file).
<b>ENAMETOOLONG</b>	The <b>program</b> field is longer than <b>PATH_MAX</b> characters or some component of <b>program</b> is longer than <b>NAME_MAX</b> .
<b>ENOENT</b>	One or more components of the path name specified by <b>program</b> do not exist or <b>program</b> points to a NULL string.
<b>ENOEXEC</b>	The contents of the <b>program</b> file are not of the form <b>#!uaaa</b> (where <b>u</b> is an uppercase letter and <b>a</b> is an uppercase letter or a digit), or the 4-character TPF program name is not of the form <b>uaaa</b> .
<b>ENOTDIR</b>	A component of the path prefix specified by <b>program</b> is not a directory.
<b>ETPFFSNOTRDY</b>	The TPF file system has not been initialized or cannot be used by the process.

**E2BIG**

The length of the **program** string plus the length of the **parm\_data** string is greater than or equal to 4094 bytes.

## Programming Considerations

- You must have authorization to use a restricted macro to call this function.
- This function performs system resource checks that may cause the calling ECB to give up control.
- If **istream** is set to **FORK\_IS\_BALANCE**, the TPF scheduler will be called.
- ECBs may or may not be started on the same I-stream where they are created depending on the options you set and whether the scheduler is called.
- The following information is inherited by the created child process from the calling parent process:
  - Environment list
  - Real user ID (UID) and real group ID (GID)
  - Process group ID
  - Current working directory
  - File mode creation mask
  - Open file descriptors that do not have the FD\_CLOEXEC file descriptor flag set to 1. The file descriptors must also be able to be inherited, that is, the device driver must specify that file descriptors that use it can be inherited.
- The effective UID and effective GID of the child process are initialized to the effective UID and effective GID of the parent process with the following exceptions:
  - If **TPF\_FORK\_FILE** is specified and if the file specified by **program** has the **set-user-ID (S\_ISUID)** flag set, the effective user ID is set to the user ID of the owner of the file.
  - If **TPF\_FORK\_FILE** is specified and if the file specified by **program** has the **set-group-ID (S\_ISGID)** flag set, the effective group ID is set to the group ID of the owner of the file.
- The saved set-user-ID and saved set-group-ID of the child process are initialized to the same values as the effective user ID and effective group ID, respectively.
- Each time a process that has a parent process exits, a SIGCHLD signal is sent to the parent process. Unless the parent process has set its SIGCHLD signal disposition to SIG\_IGN using the `signal` function, termination status for each exiting child process is saved until the parent process calls the `wait` or `waitpid` function.

## Examples

The following example creates an asynchronous entry and adds it to the ready list for the application I-stream that is least active. No data is passed to the EBW work area of the asynchronous entry. When the asynchronous entry gets control of the selected I-stream, it enters the TPF program segment specified by `"/bin/usr/usr1/app1.exe"`, which contains the string `#!QZZ2`. The program-defined environment variable `MYPATH` is passed from `QZZ1` to `QZZ2`.

```

/*****
/* Sample program QZZ1
/*****
#define TPF_FORK_EXTENDED
#include <sysapi.h>

int main(int argc, char **argv) {

    char *args[] = { "/bin/usr/usr1/app1.exe", "0", "0", "100", "ADD", NULL };

```

## tpf\_fork

```
char *envp[] = {"MYPATH=/usr/bin:/bin", NULL };
struct tpf_fork_input create_parameters;
pid_t child_pid;

create_parameters.program = "/bin/usr/usr1/app1.exe"
/*****
/*   call program QZZ2 specified by   */
/*   "/bin/usr/usr1/app1.exe"         */
*****/
create_parameters.prog_type = TPF_FORK_FILE;
create_parameters.istream = TPF_FORK_IS_BALANCE;
create_parameters.ebw_data_length = 0;
create_parameters.ebw_data = NULL;
create_parameters.parm_data = NULL;
child_pid = tpf_fork(&create_parameters, (const char **)args, (const char **)envp);
}

/*****
/* End of QZZ1                               */
*****/

/*****
/* Sample program QZZ2                               */
*****/

#include <sysapi.h>

int main(int argc, char **argv)
{
    int i;

    for (i=0; i<=argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
}

/*****
/* End of QZZ1                               */
*****/
```

The output of QZZ2 is:

```
argv[0]=/bin/usr/usr1/app1.exe"
argv[1]=0
argv[2]=0
argv[3]=100
argv[4]=ADD
```

## Related Information

- “alarm—Schedule an Alarm” on page 13
- “kill—Send a Signal to a Process” on page 298
- “raise—Raise Condition” on page 406
- “signal—Install Signal Handler” on page 495
- “tpf\_process\_signals—Process Outstanding Signals” on page 622
- “wait—Wait for Status Information from a Child Process” on page 684
- “waitpid—Obtain Status Information from a Child Process” on page 687

See the ZFILE export command in *TPF Operations* for more information about the PATH environment variable.

## tpf\_genlc—Generate a Data List

This function is used to generate data item lists to be used with event-type or sipcc function requests. A data item list is used to identify a list of items associated with a particular event or a list of destination processor ordinal numbers.

### Format

```
#include <tpfapi.h>
int tpf_genlc(void *list_ptr,
               enum t_genlc_fmt type,
               enum t_genlc_fmt func,
               enum t_genlc_opts data,
               enum t_genlc_opts options,
               void *data_ptr,
               unsigned short count,
               unsigned short size,
               long int retvalue);
```

#### list\_ptr

A pointer to the storage area where the data item list is to be returned. The specified value of the **type** parameter determines if the storage area is either a struct tpf\_ev0bk\_list\_data defined area or a user-defined area.

#### type

Specifies whether the data item list is defined by struct tpf\_ev0bk\_list\_data or is user-defined. Valid types are:

##### TPF\_GENLC\_BLOCK

Indicates that the area pointed to by the **list\_ptr** parameter is defined as struct tpf\_ev0bk\_list\_data. The data list is returned at location list\_ptr->evnblki.

##### TPF\_GENLC\_AREA

Indicates that the area pointed to by the **list\_ptr** parameter is a user-defined area. The data list is returned at the start of the user-defined data area.

#### func

Specifies the operation to perform. Valid operations are:

##### TPF\_GENLC\_CREATE

Create a new data item list by using the parameters specified.

##### TPF\_GENLC\_MODIFY

Modify an existing data item list by using parameters specified.

#### data

Specifies the type of data contained in the data items. Valid types are:

##### TPF\_GENLC\_PROCESSOR

Indicates that the data items are processor ordinal numbers as defined in processor ID table struct p1ldt in C header file c\$p1ldt. The **count** and **size** parameters are not valid when this data type is specified. The count is determined by the option specified for the **options** parameter and the size is fixed.

##### TPF\_GENLC\_POINTER

Specifies the address where data items can be found. This data type requires that you also specify the **count** and **size** parameters.

## tpf\_genlc

### options

Specifies how the data items identified by the **data** parameter are to be processed against the data item list pointed to by the **list\_ptr** parameter. Valid options are:

#### **TPF\_GENLC\_INCLUDE\_ALL**

Include or add all of the specified data items to the list.

#### **TPF\_GENLC\_INCLUDE\_ACTIVE**

Include or add only active processor ordinal numbers to the data list.

#### **TPF\_GENLC\_INCLUDE\_INACTIVE**

Include or add only inactive processor ordinal numbers to the data list.

#### **TPF\_GENLC\_EXCLUDE\_ALL**

Exclude or remove all of the specified data items from the list.

#### **TPF\_GENLC\_EXCLUDE\_ACTIVE**

Exclude or remove only active processor ordinal numbers from the data list.

#### **TPF\_GENLC\_EXCLUDE\_INACTIVE**

Exclude or remove only inactive processor ordinal numbers from the data list.

**Note:** The options that contain **\_ACTIVE** or **\_INACTIVE** are valid only when you specify **TPF\_GENLC\_PROCESSOR** for the **data** parameter. These options do not include or exclude the originating processor.

### data\_ptr

A pointer to the address where data items can be found when you specify **TPF\_GENLC\_POINTER** for the **data** parameter; otherwise, specify a value of 0.

### count

Specifies the count of input data items at the location specified in the **data\_ptr** parameter when you specify **TPF\_GENLC\_POINTER** for the **data** parameter; otherwise, specify a value of 0.

### size

Specifies the size, from 1 to 250 bytes, of input data items at the location specified in the **data\_ptr** parameter when you specify **TPF\_GENLC\_POINTER** for the **data** parameter; otherwise, specify a value of 0.

### retvalue

Specifies a 4-byte return value to be built into the specified list items. This parameter is valid only when you specify **TPF\_GENLC\_POINTER** for the **data** parameter. If a return value is indicated, it is built into one or more list data items as specified in the **count** parameter.

## Normal Return

An integer value of 0.

- If you specify **TPF\_GENLC\_BLOCK** for the **type** parameter, the data items are returned in struct `tpf_ev0bk_list_data` at `list_ptr->evnbkli`.
- If you specify **TPF\_GENLC\_AREA** for the **type** parameter, the data list is returned at the beginning of the **list\_ptr** parameter.

## Error Return

If unsuccessful, there is no return to the application. The system ends the application and a system error dump occurs.

## Programming Considerations

- This function uses information from processor ID table struct pildt when working with processor lists.
- The ev0bk and tpf\_ev0bk\_data\_list structures describe the user area that is used as a parameter to the following C functions:
  - evnqc
  - evntc
  - evnwc
  - postc
  - tpf\_genlc
  - tpf\_sawnc.

## Examples

The following example builds a processor data list in the event block that consists of only active processors.

```
#include <tpfeq.h>
#include <tpfapi.h>
struct tpf_ev0bk_list_data    event_blk;
enum t_evn_typ               event_type = EVENT_LIST;
char                         caller_provided_name;
int                         event_timeout;
enum t_state                 event_state;

enum t_genlc_fmt             func = TPF_GENLC_CREATE;
enum t_genlc_fmt             type = TPF_GENLC_BLOCK;
enum t_genlc_data            data = TPF_GENLC_PROCESSOR;
enum t_genlc_data            options = TPF_GENLC_INCLUDE_ALL;
:

tpf_genlc(&event_blk,type,func,data, options,0,0,0,0) ;
evntc(&event_blk, event_type, caller_provided_name, event_timeout
      event_state) ;
evnwc(&event_blk, event_type) ;
:
```

## Related Information

- “evnqc—Query Event Status” on page 109
- “evntc—Define an Internal Event” on page 111
- “evnwc—Wait for Event Completion” on page 113
- “postc—Mark Event Element Completion” on page 395
- “sipcc—System Interprocessor Communication” on page 504
- “tpf\_sawnc—Wait for Event Completion with Signal Awareness” on page 627.

## tpf\_gsvac—Convert an EVM Address to an SVM Address

This function converts a specified ECB virtual memory address (EVA) in the specified ECB virtual memory (EVM) to the corresponding system virtual memory (SVM) address.

### Format

```
#include <sysapi.h>
void * tpf_gsvac(const void *ecbaddr, void *evaaddr);
```

#### ecbaddr

A pointer to the target EVM that determines the segment and page tables that will be used for conversion. This is the SVM address of the target ECB.

#### evaaddr

A pointer to the EVM address to be converted.

### Normal Return

The converted address is returned to the calling function.

### Error Return

A NULL pointer is returned when the entry control block (ECB) or EVM address is incorrect.

### Programming Considerations

- Because the requesting program may not be able to directly use the SVM address that is returned by the `tpf_gsvac` function, use this function with the `tpf_movec` function.
- The target ECB and the processing ECB must be on the same I-stream. The system virtual address (SVA) that is returned is valid for the SVM of the I-stream on which the program is active.
- When using the SVM address that is returned by the `tpf_gsvac` function, do not attempt to cross a page boundary without calling another `tpf_gsvac` function. Converting the EVM address for the next page will avoid errors caused by discontinuous SVM storage.

### Examples

The following example converts an EVM address to an SVM address.

```
#include <sysapi.h>
sva_address = (void *) tpf_gsvac((void *) g->appl_ecb, (void *)
                                ((uint) applEVA + length - 1));
if (!sva_address) return RC_ENDADDR;
sva_address = (void *) tpf_gsvac((void *) g->appl_ecb, applEVA);
if (!sva_address) return RC_STARTADDR;
tpf_movec(sva_address, TPF_MOVEC_SVA, buffer, TPF_MOVEC_EVA, length);
return RC_OK;
```

### Related Information

“`tpf_movec`—Move Data between EVM and SVM” on page 616.



## tpf\_help—Issue Help Messages

This function provides an easy way to send you help messages for particular parameters. You do not need to write any C code for this function; it only requires that you provide data structures to issue the correct help message.

### Format

```
#include <c$help.h>
```

```
void tpf_help(const char *input_message_pointer,
              const char *help_syntax_pointer,
              TPF_HELP *main_help_structure_pointer,
              const char *prefix_pointer);
```

#### **input\_message\_pointer**

Specifies the address of the input message for which a help message must be issued. HELP or ? at the beginning of the message is skipped and the remainder of the message is parsed with the IPRSE\_parse function.

#### **help\_syntax\_pointer**

Specifies the address of the syntax that is used in the IPRSE\_parse function to split the input message into its parameters.

#### **main\_help\_structure\_pointer**

Specifies the address of the help structure array that you must create. The last entry in the structure must be NULL terminated.

The first entry in the structure is used for a simple help display; for example, this message is displayed whenever an entry is called with the help request without any additional parameters. This message is also displayed when the parser returns with an error.

#### **prefix\_pointer**

Prefixes the help message, which has a severity code of I and a number of 0.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

The TPF\_HELP data structure has several fields that you must initialize, as follows:

- **action**

This field contains the address of the action code as specified in the syntax string; for example, ACTivate would be valid. Set this field to NULL to mark the end of the help structure.

- **text\_ptr**

This pointer specifies the address of a string. The string can contain \n to continue on a new line. The maximum length of a line is restricted to 255 characters; the string itself is not limited in any way.

- **sub\_help**

When the action code specified previously has additional subparameters, this field can contain the address of another TPF\_HELP array.

## tpf\_help

Because many definitions are required for this function, call this function in a subfunction that is called from the main function. To best handle the help requests, store the definitions in their own source file.

## Examples

The following example shows how the tpf\_help function is used.

```
#include <tpfapi.h>
#include <tpfparse.h>
#include <c$help.h>
#include <stdlib.h>

void my_help(char *input_ptr)
{
    const char oth_message[] =
        "This is the last help text which is available to show\n"
        "the nice and easy use of the tpf_help function. Please come\n"
        "again.";

    TPF_HELP me_sub_help[] = {
        { "OTHERS", oth_message, NULL },
        { NULL, NULL, NULL }
    };

    const char all_message[] =
        "Again a short help message";

    const char me_message[] =
        "This parameter has a subparameter and you can have\n"
        "some additional help when you specify the following:"
        "  OTHERS";

    const char non_message[] =
        "None, but still a display";

    TPF_HELP dis_sub_help[] = {
        { "ALL", all_message, NULL },
        { "ME", me_message, me_sub_help },
        { "NONE", non_message, NULL },
        { NULL, NULL, NULL }
    };

    const char main_help_message[] =
        "Welcome to the wonders of the tpf_help function\n"
        "This string has a carriage return in it and will\n"
        "write four lines with just one single wtopc call,\n"
        "which is more efficient than doing it for each line.\n"
        "Please notice that only strings with a ',' at the end\n"
        "will have a pointer and do not need to have a\n"
        "carriage return at the end. But now, I will show you\n"
        "some more help if you select one of the following:"
        "  ADD\n  ACTivate\n  DEActivate\n  DElete\n  DISplay";

    const char add_message[] =
        "This is an explanation of the ADD Parameter since you\n"
        "requested additional help on this."
        "There are no subparameters for the ADD function and\n"
        "the function makes no sense anyway";

    const char act_message[] =
        "Very short help message";

    const char dea_message[] =
        "Another short help message";
```

```

const char del_message[] =
    "The last short help message";

const char dis_message[] =
    "Now this is a bigger help display, because we can select\n"
    "additional help for three subparameters.  "
    "Request help for one of the following parameters :\n"
    "  ALL\n  ME\n  NONE";

TPF_HELP help[] = {
    { "HELP",main_help_message,NULL},      /* <- This is default */
    { "ADD",add_message,NULL},
    { "ACTivate",act_message,NULL},
    { "DEActivate",dea_message,NULL},
    { "DElete",del_message,NULL},
    { "DISplay,dis_message,dis_sub_help},
    { NULL,NULL,NULL}
};

const char *syntax =
    " { "
    "   " ADD "
    " | "
    "   " ACTivate "
    " | "
    "   " DEActivate "
    " | "
    "   " DElete "
    " | "
    "   " DISplay "
    " [ "
    "   { "
    "     " ALL "
    "     | "
    "     " ME [ OTHERS ] "
    "     | "
    "     " NONE "
    "   } "
    " ] "
    " } ";

    tpf_help(input_ptr,syntax,help,"TEST");
}

```

```

int main()
{
    const char *syntax =
        " { "
        "   " ADD "
        " | "
        "   " ACTivate "
        " | "
        "   " DEActivate "
        " | "
        "   " DElete "
        " | "
        "   " DISplay "
        " { "
        "     " ALL "
        "     | "
        "     " ME [ OTHERS ] "
        "     | "
        "     " NONE "
        "   } "

```

## tpf\_help

```
" } ";

struct mi0mi *block_ptr;
char *input_ptr;
struct IPRSE_output parsed_struct;
struct IPRSE_output *next_ptr;
long result;

block_ptr=(struct mi0mi *)ecbptr()->celcr0;
input_ptr=block_ptr->mi0acc;
input_ptr=input_ptr+5;

result=IPRSE_parse(input_ptr,syntax,&parsed_struct,
                  IPRSE_EOM+IPRSE_PRINT+IPRSE_ALLOC,
                  "TEST");

/*****
 * Any error? IPRSE_BAD Message already send
 *****/

if (result!=IPRSE_BAD) {

    /*****
     * help request ?
     *****/

    if (result==IPRSE_HELP) {

        my_help(input_ptr);

    } else {

        .
        .
        .
        .
    }
}
exit(0);
}
```

## Related Information

- “IPRSE\_parse—Parse a Text String against a Grammar” on page 1456
- “wtopc—Send System Message” on page 700.

## tpf\_is\_RPCServer\_auto\_restarted—Query If an RPC Server Is Restarted

This function queries a remote procedure call (RPC) server to see if it has been restarted automatically by the TPF 4.1 system. Whenever a loadset is activated, deactivated, or excluded by the E-type loader, all RPC servers are restarted automatically.

### Format

```
#include <rpc.h>
void tpf_is_RPCServer_auto_restarted();
```

### Normal Return

The `tpf_is_RPCServer_auto_restarted` function returns one of the following:

- 0** This is the original server that was started from the Internet daemon (INETD).
- 1** This is a copy of the original version of the server that was restarted automatically by the TPF 4.1 system.

### Error Return

Not applicable.

### Programming Considerations

Use this application programming interface (API) only for RPC server applications.

### Examples

The following example queries the start status of the RPC server.

```
void rpcsrvstrt(void) {

    /******
    /* If this is the original version of the RPC server, */
    /* then perform server initialization code.          */
    /******

    #include <rpc.h>

    if (!tpf_is_RPCServer_auto_restarted()) {
        server_init_fct1();
        server_init_fct2();
        server_init_fct3();
    }
    else
        printf("RPC server is restarted by TPF\n");
}
```

### Related Information

None.

## tpf\_itrpc–Send Simple Network Management Protocol User Trap

This function sends a Simple Network Management Protocol (SNMP) enterprise-specific trap to the remote SNMP destination managers specified in the `/etc/snmp.cfg` SNMP configuration file.

### Format

```
#include <c$snmp.h>
int tpf_itrpc(char* varlist, int varlen, int spectrap);
```

#### varlist

A pointer to the variable binding list associated with the SNMP enterprise-specific trap being generated.

#### varlen

The length, in bytes, of the list specified with **varlist**. If there is no variable binding list, set this parameter to 0.

#### spectrap

An implementation-specific value for the SNMP enterprise-specific trap that describes the trap being generated.

### Normal Return

Table 20. *tpf\_itrpc* Normal Return

Value Name	Return Code	Description
ISNMP_ITRPC_OK	0	An SNMP enterprise-specific trap was successfully built and queued to be sent to the network.

### Error Return

If an attempt to build an SNMP enterprise-specific trap is unsuccessful, you will receive one of the following error returns:

Table 21. *tpf\_itrpc* Error Return

Value Name	Return Code	Description
ISNMP_ITRPC_STATE_ERROR	–1	The TPF system is not in CRAS state or higher.
ISNMP_ITRPC_CONFIG_ERROR	–2	The <code>/etc/snmp.cfg</code> SNMP configuration file was not refreshed successfully.  Do the following: 1. Create or fix the <code>/etc/snmp.cfg</code> SNMP configuration file. 2. Enter the ZSNMP command with the REFRESH parameter specified. 3. Issue the <code>tpf_itrpc</code> function again.
ISNMP_ITRPC_NO_TRAPS	–3	The <code>/etc/snmp.cfg</code> SNMP configuration file specifies that the TRAPIP keyword is coded as NONE, disabling SNMP trap support.  If you want to send traps, code the TRAPIP keyword in the <code>/etc/snmp.cfg</code> SNMP configuration file with a valid SNMP destination manager IP address or host name.
ISNMP_ITRPC_ENCODE_ERROR	–4	An error occurred when encoding one of the trap values.

Table 21. *tpf\_itrpc* Error Return (continued)

Value Name	Return Code	Description
ISNMP_ITRPC_SIZE_ERROR	-5	The size of the trap exceeded the maximum message size of 548 bytes.

## Programming Considerations

- The TPF system must be in CRAS state or higher to use this function.
- The variable binding list must be in the correct SNMP basic encoding rules (BER) format. For information about BER encoding, see ISO 8825 *Part 1: Basic Encoding Rules*. Go to <http://www.iso.ch/> to view ISO 8825.
- Ensure that the `/etc/snmp.cfg` SNMP configuration file has been refreshed successfully and that the remote SNMP managers are defined to it by entering the ZSNMP command. See *TPF Operations* for information about the ZSNMP command.
- Ensure TCP/IP native stack support is defined to the TPF system. See *TPF Transmission Control Protocol/Internet Protocol* for information about SNMP agent support and TCP/IP native stack support.
- This function is implemented in dynamic link library (DLL) CNMP. You must use the definition side-deck for DLL CNMP to link-edit an application that uses this function.

## Examples

The following example sends the enterprise-specific trap defined in **spectrap**, and containing the variable binding list pointed to by **encoded\_list**, to all the SNMP destination managers specified in the `/etc/snmp.cfg` SNMP configuration file.

```

/*****
/*          Include files          */
/*****
#include <c$snmp.h>
#include <stdlib.h>
#include <string.h>
/*****
/*          #DEFINES          */
/*****
#define SPECTRAP      4

/*****
/*****
extern "C" void QZZ2()
{

    int          varlen = 0, spectrap = 0, rc;
    char          object_id[24] = "\x2b\x6\x1\x4\x1\x1";
    char          *temp;
    char          encoded_var[100];
    int          encoded_var_len = 0;
    char          encoded_bind[100];
    int          encoded_bind_len;
    char          encoded_list[100];
    int          encoded_list_len;
    int          value=100;
    struct        snmp_struct  encode_struct;

    temp = encoded_var;

/*  Encode the OBJECT ID for the Variable Binding */

```

## tpf\_itrpc

```
encode_struct.snmp_input_value = object_id
encode_struct.snmp_input_length = strlen(object_id);
encode_struct.snmp_input_type = ISNMP_TYPE_OBJECTID;
encode_struct.snmp_output_value = encoded_var;

    if (tpf_snmp_BER_encode(&encode_struct) != 0)
        exit(0);

encoded_var_len += encode_struct.snmp_output_length;
temp += encode_struct.snmp_output_length;

/* Encode the value and copy the encoded value after the */
/* OBJECT ID. Increment the size of the VAR-BIND          */
/*

encode_struct.snmp_input_value = &value;
encode_struct.snmp_input_length = sizeof(int);
encode_struct.snmp_input_type = ISNMP_TYPE_INTEGER;
encode_struct.snmp_output_value = temp;

    if (tpf_snmp_BER_encode(&encode_struct) != 0)
        exit(0);

encoded_var_len = encode_struct.snmp_output_length;

/* Encode the variable bind as a SEQUENCE OF              */
/*

encode_struct.snmp_input_value = encoded_var;
encode_struct.snmp_input_length = encoded_var_len;
encode_struct.snmp_input_type = ISNMP_TYPE_SEQUENCE_OF;
encode_struct.snmp_output_value = encoded_bind;

    if (tpf_snmp_BER_encode(&encode_struct) != 0)
        exit(0);

encoded_bind_len = encode_struct.snmp_output_length;

/* Variable Binding complete. Encode the entire          */
/* variable binding list as a sequence of                  */
/*

encode_struct.snmp_input_value = encoded_bind;
encode_struct.snmp_input_length = encoded_bind_len;
encode_struct.snmp_input_type = ISNMP_TYPE_SEQUENCE_OF;
encode_struct.snmp_output_value = encoded_list;

    if (tpf_snmp_BER_encode(&encode_struct) != 0)
        exit(0);

encoded_list_len = encode_struct.snmp_output_length;

/* Fill in the specific TRAP information and call         */
/* tpf_itrpc passing the variable binding list and         */
/* length                                                  */
/*

spectrap = SPECTRAP;

rc = tpf_itrpc(encoded_list, encoded_list_len, spectrap);

    exit(0);
}
```



## **Related Information**

“tpf\_snmp\_BER\_encode—Encode SNMP Variables in BER Format” on page 632.

## tpf\_lemic –Lock Entry Management Interface

This function issues one or more service requests to a coupling facility (CF) that is being used as an external locking facility (XLF).

### Format

```
#include <c$cf1p.h>
int      tpf_lemic(enum lemic_req req,
                  struct icf1p *cf1p_ptr);
```

#### req

A constant value representing the type of service request being performed:

##### LEMIC\_CLEARUSER

Clears all locks, any held lock requests, or any queued lock requests for a particular user from a CF.

##### LEMIC\_DELETE

Removes one or more held locks from a CF.

##### LEMIC\_DELETESET

Removes a set of locks associated with a particular CF list structure from a CF.

##### LEMIC\_LOCATELOCK

Locates a CF and list number where each lock resides for one or more lock names.

##### LEMIC\_LOCATEMOD

Locates a CF and the starting and ending list numbers used for locking by a particular module.

##### LEMIC\_MANAGE

Sends one or more set, release, or withdraw lock commands, in any combination, to a CF.

##### LEMIC\_MONITOR

Registers a user for lock granted and contention notification.

##### LEMIC\_READ

Reads one or more locks in a CF.

##### LEMIC\_READUSER

Reads all locks held by a particular user on a CF.

##### LEMIC\_VERIFYCF

Verifies the connectivity between a processor and a CF.

#### cf1p\_ptr

A pointer to the block of storage that is mapped by the c\$cf1p.h header and must reside in a single page, and that is passed to the service routine.

### Normal Return

#### COMPLETED\_OK

The function is completed successfully and control is returned to the next sequential instruction (NSI).

### Error Return

#### ICFLP\_SYSERR

An error condition occurred in the CCCFLC service routines.

Exit the application. You can also examine the console logs to find any associated system error dumps.

If this return code is received, the ICFLP\_RSC field contains one of the following response codes:

<b>TIMEOUT</b>	<p>The requested operation was not able to be completed because a timeout occurred.</p> <p>Submit the operation again using the CF locking restart token provided in ICFLP_RT.</p>
<b>INV_RESTOK</b>	<p>A program error occurred because a CF locking restart token that is not valid was provided for an operation that was resubmitted.</p> <p>Do the following:</p> <ol style="list-style-type: none"> <li>1. Correct the CF locking restart token.</li> <li>2. Try the tpf_lemic function again.</li> </ol>
<b>DB_FILLED</b>	<p>The data block provided with either a <b>LEMIC_READ</b> or <b>LEMIC_READUSER</b> operation is filled and there are more locks to read.</p> <p>Submit the <b>LEMIC_READ</b> or <b>LEMIC_READUSER</b> operation again using the CF locking restart token provided in ICFLP_RT and a data block that no longer contains critical data.</p>
<b>LN_MM</b>	<p>A list number mismatch occurred.</p> <p>Do the following:</p> <ol style="list-style-type: none"> <li>1. Submit a new operation (such as <b>LEMIC_LOCATELOCK</b>, <b>LEMIC_LOCATEMOD</b>, <b>LEMIC_READ</b>, or <b>LEMIC_READUSER</b>) that will return a valid list number.</li> <li>2. Submit the original operation again using a valid list number.</li> </ol>
<b>LN_DNX</b>	<p>The list number does not exist.</p> <p>Do the following:</p> <ol style="list-style-type: none"> <li>1. Submit a new operation (such as <b>LEMIC_LOCATELOCK</b>, <b>LEMIC_LOCATEMOD</b>, <b>LEMIC_READ</b>, or <b>LEMIC_READUSER</b>) that will return a valid list number.</li> <li>2. Submit the original operation again using a valid list number.</li> </ol>
<b>ICFLP_SYSERR</b>	<p>An error condition occurred in the CCCFLC service routines.</p> <p>Exit the application. Additionally, you can examine the console logs to find any associated system error dumps.</p>
<b>INS_SPACE</b>	<p>An error occurred during a read operation because there is not enough space in the data block.</p>
<b>GLBL_MM</b>	<p>A global lock manager mismatch occurred.</p>

	Exit the application. If necessary, see your IBM service representative for more information.
<b>LCL_MM</b>	A lock manager mismatch occurred.  Exit the application. If necessary, see your IBM service representative for more information.
<b>LAU_MM</b>	A list authority mismatch error occurred.  Exit the application. If necessary, see your IBM service representative for more information.
<b>INS_MB_SPACE</b>	A program error occurred because there is not enough message buffer space available to read the locks in a CF or to read all locks held by a particular user on a CF.  Exit the application. If necessary, see your IBM service representative for more information.
<b>ABNORMAL</b>	A program error occurred because of an abnormal locking condition. Some of the set, release, or withdraw commands were not processed by a CF.  Exit the application. If necessary, see your IBM service representative for more information.
<b>RRC_ERROR</b>	An error occurred while processing a monitor operation.  Exit the application. If necessary, see your IBM service representative for more information.
<b>CVCF_ERROR</b>	The CCCFLC service routines were unable to determine the CF or the list number from the information provided with the operation.  Exit the application and review the application code to ensure the operation was set up correctly. If necessary, see your IBM service representative for more information.
<b>CMD_SUPPR</b>	A program error occurred because the function call was suppressed.  Exit the application and review the console logs to determine if the CF specified in the operation was added correctly to the CF locking configuration. If necessary, see your IBM service representative for more information.
<b>CMD_TERM</b>	A program error occurred because the function call ended.

## Programming Considerations

- A waitc call is issued implicitly with this function.
- **Attention:** Using this function can cause corruption of the locking control information on the CF. This corruption can lead to errors that cannot be predicted and online database integrity problems.
- The calling program returns storage when the storage is no longer needed.
- If the specified CF or CF locking structure is not valid, a system error is issued.
- See the c\$cf1p.h header to determine the required input fields and the information included on return in the ICFLP\_DATA control area.

- The reply code for individual requests must be checked to determine the success or failure of that specific request.

## Examples

The following example shows the setup and function call for the **LEMIC\_CLEARUSER** operation.

```

cflp_ptr = getcc( D2, GETCC_TYPE+GETCC_FILL, L4, 0x00 );

memcpy( &cflp_ptr->icflp_ctl.icflp_id, "CFLP",
        sizeof( cflp_ptr->icflp_ctl.icflp_id ));

cflp_ptr->icflp_ctl.icflp_condata.icflp_cfix = cf_index;
cflp_ptr->icflp_ctl.icflp_condata.icflp_stix =
    (unsigned char)CFL_LOCKSPACE_STRUCTURE_INDEX;
cflp_ptr->icflp_ctl.icflp_condata.icflp_uid =
    cflt_ptr->icflt_hdrc.icflt_cflgs.icflt_uid;

rc = tpf_lemic( LEMIC_CLEARUSER, cflp_ptr );

```

## Related Information

None.

## tpf\_mvec—Move Data between EVM and SVM

This function moves data between an ECB virtual memory (EVM) address space and a system virtual memory (SVM) address space. `tpf_mvec` permits an ECB-controlled program to read and modify storage that is not part of its own address space.

### Format

```
#include <sysapi.h>
void tpf_mvec(const void *fromaddr, int fromvm, void *toaddr,
              int tovm, long length);
```

#### **fromaddr**

Specifies the location from which data will be moved.

#### **fromvm**

Specifies whether the location from which data is being moved is an EVM address space or an SVM address space. The following symbols are defined in the `sysapi.h` header file for use in the **fromaddr** member:

- `TPF_MOVEEC_EVA` - Identifies the address as an EVA address.
- `TPF_MOVEEC_SVA` - Identifies the address as an SVA address.

#### **toaddr**

Specifies the location to which data will be moved.

#### **tovm**

Specifies whether the location to which data is being moved is an EVM address space or an SVM address space. The following symbols are defined in the `sysapi.h` header file for use in the **fromaddr** member:

- `TPF_MOVEEC_EVA` - Identifies the address as an EVA address.
- `TPF_MOVEEC_SVA` - Identifies the address as an SVA address.

#### **length**

Specifies the length of the data to be moved.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- This function requires that the application program allocation specify key 0 write authorization and authorization to issue a restricted macro.
- This function may be executed on any I-stream.
- You can move data between address spaces as follows:
  - From EVM to SVM
  - From SVM to EVM
  - From EVM to EVM
  - From SVM to SVM.

**Note:** The EVM-to-EVM combination is restricted to movement in a single ECB virtual memory, not between two EVMs. To move data from one EVM to another EVM, use the `tpf_mvec_EVM` function.

- System error dumps can occur when servicing a tpf\_movec request. See *TPF Messages, Volume 1* and *TPF Messages, Volume 2* for more information.

## Examples

The following example shows an ECB field being moved from an SVM address space to an EVM address space:

```
#include <sysapi.h>
:
struct eb0eb * sva = usoEntry.sva_address;
u_long ecbActive = 0;
tpf_movec(&sva->celaii, TPF_MOVEC_SVA,
          &ecbActive,  TPF_MOVEC_EVA,
          sizeof(ecbActive));
```

## Related Information

“tpf\_movec\_EVM—Move Data from One EVM to Another EVM” on page 618.

---

## tpf\_movec\_EVM—Move Data from One EVM to Another EVM

This function moves data from one ECB virtual memory (EVM) address space to another EVM address space. The tpf\_movec\_EVM function permits an ECB-controlled program to read and modify storage that is not part of its own address space.

### Format

```
#include <sysapi.h>
void tpf_movec_EVM(const void *fromaddr, int fromecb, void *toaddr,
                  int toecb, long length);
```

**fromaddr**

Specifies the ECB virtual address (EVA) from which data will be moved.

**fromecb**

Specifies the system virtual address (SVA) of the entry control block (ECB) from which data is moved.

**toaddr**

Specifies the EVA to which data will be moved.

**toecb**

Specifies the SVA of the ECB to which data is moved.

**length**

Specifies the length of the data to be moved.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- This function requires that the application program allocation specify key 0 write authorization and authorization to issue a restricted macro.
- This function can be run on any I-stream.
- This function moves data from one EVM to another EVM. If you want to move data between an EVM and the SVM, use the tpf\_movec function instead.
- System error dumps can occur when servicing a tpf\_movec\_EVM request. See *TPF Messages, Volume 1* and *TPF Messages, Volume 2* for more information.

### Examples

An example is not provided because ECB addresses for the TPF system vary from customer site to customer site, making it difficult to provide a meaningful example.

### Related Information

“tpf\_movec—Move Data between EVM and SVM” on page 616.

See *TPF System Macros* for information about the \$MOVEC macro.



## tpf\_msg—Issue a Message

This function sends informational or error messages to the terminal. The message is provided as a structure containing the message number, severity code, routing information, and message text.

### Format

```
#include <c$mesg.h>
```

```
void tpf_msg(const char *prefix, TPF_MSG *msg, va_list arg_ptr)
```

#### prefix

A 4-byte character string used for prefixing the output message.

#### msg

A pointer to the TPF\_MSG structure to be used.

#### arg\_ptr

An argument pointer list provided by the `va_start` function. It provides an undefined number of arguments.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The message is written using the `wtopc` function. When the specified message text is greater than 255 bytes (a `wtopc` function restriction), the `tpf_msg` function will split it and use the chain option to send the message.
- The output from each message contains a `wtopc` header, including the time.
- The message must be defined in a TPF\_MSG structure, as follows:

```
typedef const struct {
    const unsigned short number;
    const unsigned char severity;
    const long rout_list;
    const char * const text;
} TPF_MSG;
```

The fields are the same type used for the `wtopc` function:

#### – Number

This is the message number used in the header. This number must be unique for every message in the package.

#### – Severity

The severity code, which identifies the type of message. See “`wtopc_insert_header`—Save Header for `wtopc`” on page 704 for a list of predefined values.

#### – rout\_list

The routing list, which specifies the destination of the message. You can specify more than one type, in which case the message is copied to both devices. See “`wtopc`—Send System Message” on page 700 for a list of routing destinations.

#### – Text

## tpf\_msg

This is the output message, which is one constant string that may be longer than 255 bytes. It can contain any format attribute that is supported by the `vsprintf` function.

- In low storage situations, it is possible that the function will not be able to allocate a buffer to format the output message. When this occurs, the input message is written to the terminal in the original format.

## Examples

The following example shows how the `tpf_msg` function is used. The message handler (which is `send_message()` in this example) must be in a separate function to reduce system overhead.

```
#include <tpfapi.h>
#include <c$mesg.h>
.
.
/*      POSIX Note                                */
/*The following defines should be done in the c$mesg.h header file. */
/*The prefix can be a symbolic name such as OLDR or                */
/*the program name from the caller. A                               */
/*symbolic name is easier to handle in an automated                */
/*environment than the program name; however, the                  */
/*program name provides fast access to the segment                 */
/*where the message originated. A possible solution to             */
/*meet both requirements is shown in MSG_THREE.                    */
/*                                                                    */

#define MSG_ONE      1
#define MSG_TWO      2
#define MSG_THREE    3
#define MSG_FOUR     4
#define FCT_PFX      "OLDR"
:
:
void send_message(const char *prefix,long nbr, ...)
{
    TPF_MSG msg[] = {
        {
            0,WTOPC_ERROR,WTOPC_EBROUT,
            "INVALID MESSAGE NUMBER %d PASSED TO FUNCTION"
        },
        {
            MSG_ONE,WTOPC_INFO,WTOPC_EBROUT+WTOPC_RO,
            "INFO MSG FLOAT %f LONG %ld\n"
            "NEW LINE\nSTRING %s"
        },
        {
            MSG_TWO,WTOPC_ERROR,WTOPC_EBROUT,
            "SIMPLE ERROR MESSAGE"
        },
        {
            MSG_THREE,WTOPC_ERROR,WTOPC_EBROUT,
            "%s - ACTIVATE ERROR\n"
            "LOADSET %s WAS NOT AVAILABLE FOR USER %061X"
        }
    };
    va_list arg_ptr;

    if ((nbr<0) || (nbr>=(sizeof(msg)/sizeof(TPF_MSG)))) {
        send_message(prefix,0,nbr);
    } else {
        va_start(arg_ptr,nbr);
        tpf_msg(prefix,&msg[nbr],arg_ptr);
        va_end(arg_ptr);
    }
    return;
}
```

```

      .
      .
      .
long main()
{
    send_message(FCT_PFX,MSG_ONE,1.23,999,"HELLO WORLD");
    send_message(FCT_PFX,MSG_TWO);
    send_message(FCT_PFX,MSG_THREE,"PROG","S011200",terminal);
    send_message(FCT_PFX,MSG_FOUR);
    exit(0);
}
      .
      .
      .

```

**Output:**

```

OLDR0001I 12.35.48 INFO MSG FLOAT 1.230000 LONG 999
NEW LINE
STRING HELLO WORLD
OLDR0002E 12.35.48 SIMPLE ERROR MESSAGE
OLDR0003E 12.35.48 PROG - ACTIVATE ERROR
LOADSET S011200 WAS NOT AVAILABLE FOR USER 0F120E
OLDR0000E 12.35.48 INVALID MSG NUMBER 4

```

**Related Information**

- “wtopc—Send System Message” on page 700
- “vsprintf—Format and Print Data to a Buffer” on page 682.

---

## tpf\_process\_signals–Process Outstanding Signals

This function checks to see if there are any outstanding signals from another process (sent by the kill function) and performs a raise function to handle each outstanding signal.

### Format

```
#include <sysapi.h>
int tpf_process_signals(void)
```

### Normal Return

The tpf\_process\_signals function is always successful; it returns the number of signals processed.

**Note:** The handling of some signals can cause the calling entry control block (ECB) to exit.

### Error Return

Not applicable.

### Programming Considerations

- If the signal function was used to install signal handlers, the placement of the tpf\_process\_signals function in the code is subject to the same restrictions as the raise function.

### Examples

The following example shows a program checking for outstanding signals in a loop that creates child processes.

```
#include <sysapi.h>
:
:
struct tpf_fork_input create_parameters;
pid_t child_pid;

while (!exit_flag) {
create_parameters.program = "/bin/usr/usr1/app1.exe"
:
child_pid = tpf_fork(&create_parameters);

tpf_process_signals();
}
```

### Related Information

- “alarm–Schedule an Alarm” on page 13
- “kill–Send a Signal to a Process” on page 298
- “sigaction–Examine and Change Signal Action” on page 486
- “signal–Install Signal Handler” on page 495
- “sigpending–Examine Pending Signals” on page 498
- “sigprocmask–Examine and Change Blocked Signals” on page 499
- “sigsuspend–Set Signal Mask and Wait for a Signal” on page 502
- “sleep–Suspend the Calling Process” on page 507
- “tpf\_fork–Create a Child Process” on page 594
- “wait–Wait for Status Information from a Child Process” on page 684

- “waitpid—Obtain Status Information from a Child Process” on page 687.

## tpf\_rcrfc—Release a Core Block and File Address

This function combines the `relcc` and `relfc` functions. The storage block specified by the designated core block reference word (CBRW) of an entry control block (ECB) data level or a data event control block (DECB) is released from the ECB. The pool file address specified by the designated file address reference word (FARW) of an ECB data level or DECB is returned to the available pool of file records.

### Format

```
#include <tpfio.h>
tpf_rcrfc(enum t_lvl level)
```

or

```
#include <tpfio.h>
tpf_rcrfc(TPF_DECB *decb)
```

#### level

One of 16 possible values representing a valid ECB data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This indicates the CBRW containing the address of the working storage block to be returned to the system and the FARW containing the pool file address to be returned to the system.

#### decb

A pointer to a DECB. This indicates the CBRW containing the address of the working storage block to be returned to the system and the FARW containing the pool file address to be returned to the system.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- File pool support must be active when this function is issued; the system must be above UTIL (utility) state.
- This function checks the CBRW of the specified ECB data level or DECB to determine if a storage block is held. A system error dump is taken if a block is not held and the entry is exited.
- TPF transaction services processing affects `tpf_rcrfc` function processing in the following ways:
  - The file address is released only at commit time. For more information about commit time, see “`tx_commit`—Commit a Global Transaction” on page 649.
  - After the `tx_rollback` function has completed successfully, file address release requests are discarded and file addresses are not released.
  - If a system error occurs, processing ends as if the `tx_rollback` function was issued.
- Applications that call this function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.

## Examples

The following example releases a core block and file pool address from an ECB data level and a DECB.

```
#include <tpfio.h>
...
TPF_DECB *decb
...
tpf_rcrfc(D6);    /* Release block and file pool address held in D6 */
...
tpf_rcrfc(decb); /* Release block and file pool address held in DECB */
```

## Related Information

- “relcc—Release Working Storage Block” on page 421
- “relfc—Release File Pool Storage” on page 423.

See *TPF Application Programming* for more information about DECBs.

---

## tpf\_RPC\_options–Set TPF- Specific Thread Options

This function sets remote procedure call (RPC) thread options.

### Format

```
#include <rpc.h>
void tpf_RPC_options(int options);
```

#### options

One of the following, specifying whether the associated process exits when this RPC thread entry control block (ECB) exits:

##### **TPF\_RPC\_OPTIONS\_EXIT\_PROCESS**

If this RPC thread ECB exits, its associated process also exits.

##### **TPF\_RPC\_OPTIONS\_EXIT\_THREAD**

If this RPC thread ECB exits, its associated process does not exit.

### Normal Return

None.

### Error Return

None.

### Programming Considerations

This application programming interface (API) can be issued only for an RPC threaded ECB.

**Note:** If this API is issued for a non-threaded ECB, the API is ignored.

### Examples

The following example shows that the process does not exit when the ECB exits.

```
#include <rpc.h>

/*      If this ECB exits, leave the process running      */
tpf_RPC_options(TPF_RPC_OPTIONS_EXIT_THREAD);
```

### Related Information

None.



## tpf\_sawnc—Wait for Event Completion with Signal Awareness

This function waits for a named event to be completed and is similar to the `evnwc` function; however, if the caller of the `tpf_sawnc` function has to wait for an event to be completed, the caller can be interrupted by a signal. The `tpf_sawnc` function is used with the `evntc` and `postc` functions.

### Format

```
#include <tpfapi.h>
int tpf_sawnc(struct ev0bk *evninf, enum t_evn_typ type);
```

#### evninf

A pointer to the `tpf_sawnc` parameter block. See the `ev0bk` and `tpf_ev0bk_list_data` structures for more information about the `tpf_sawnc` parameter block.

#### type

The type of event being completed. The argument must belong to the enumerated type `t_evn_typ`, which is defined in the `tpfapi.h` header file. Use one of the following predefined terms:

#### EVENT\_MSK

for mask events.

#### EVENT\_CNT

for count events.

#### EVENT\_CB\_Dx

where *x* is a single hexadecimal digit (**0–F**) for core events.

#### EVENT\_LIST

for list events.

### Normal Return

An integer value of 0.

- If this is a core block event, the core block reference word (CBRW) that you specified for the **type** parameter contains the core block that was passed by the `postc` function.
- If the event is count-type, the `evninf->evnpstinf.evnbk1` structure contains the remaining count value for the event.
- If the event is list-type, the `evninf->evnbklc` structure contains the count of list items, the `evninf->evnbkls` structure contains the size of a list item, and the `evninf->evnbkli` structure contains the list items for the event.
- If the event is mask-type, the `evninf->evnpstinf.evnbk1` structure contains the remaining mask value for the event.
- The `evninf->evnbkm2` structure contains the accumulated POST MASK 2 for the event.
- The `evninf->evnbke` structure contains the error indicator if an error has occurred, or 0 if no error has occurred.

### Error Return

One of the following:

- An integer value of 1 if the event name is not found
- An integer value of 2 if the event is an error post
- An integer value of 3 if the event is interrupted by a signal.

## Programming Considerations

- This function causes a loss of control for the issuing entry control block (ECB) unless the named event is not found. The calling ECB regains control when one of the following occurs:
  - Event completion is posted (with or without error).
  - A timeout occurs.
  - A pending alarm is triggered, causing a SIGALARM signal to be sent.
  - A signal is sent to the calling process.
- More than one ECB is allowed to wait for the same named event except for core block events. When the event is completed, all waiting ECBs are posted. For a core block event, only the creating ECB can wait for the event. Any other ECB trying to wait for the event is returned with a not-found condition.

## Examples

The following example creates an event and waits for the event to be completed.

```
#include <tpfapi.h>
struct ev0bk          event_blk;
enum t_evn_typ        event_type;
char                  caller_provided_name;
int                   event_timeout;
enum t_state          event_state;

:

evntc(&event_blk, event_type, caller_provided_name, event_timeout
      event_state);
tpf_sawnc(&event_blk, event_type);

:
```

## Related Information

- “evinc—Increment Count for Event” on page 108
- “evnqc—Query Event Status” on page 109
- “evntc—Define an Internal Event” on page 111
- “evnwc—Wait for Event Completion” on page 113
- “postc—Mark Event Element Completion” on page 395
- “tpf\_genlc—Generate a Data List” on page 599.

## tpf\_select\_bsd—Indicates Read, Write, and Exception Status

This function indicates which of the specified file descriptors is ready for reading and writing, or has an error condition pending. If the specified condition is false for all of the specified file descriptors, the `tpf_select_bsd` function blocks up to the timeout interval specified, until the specified condition is true for at least one of the specified file descriptors.

### Format

```
#include <sys/time.h>
int tpf_select_bsd(int maxfds, fd_set* reads, fd_set* writes,
                  fd_set* excepts, struct timeval* tv)
```

#### **maxfds**

The value is ignored.

#### **reads**

A pointer to a set of file descriptors to be checked for readiness for reading.

#### **writes**

A pointer to a set of file descriptors to be checked for readiness for writing.

#### **excepts**

A pointer to a set of file descriptors to be checked for exception pending conditions.

**tv** A pointer to the maximum timeout interval in milliseconds.

If the **tv** argument is not NULL, it points to an object of type *struct timeval* that specifies a maximum interval to wait for the selection to complete. If the **tv** argument points to an object whose members are 0, the `tpf_select_bsd` function does not block. If the **tv** argument is a null pointer, the `tpf_select_bsd` function blocks until an event causes one of the file descriptor sets to not be empty. If the time limit expires before any event occurs that would cause one of the file descriptor sets not to be empty, the `tpf_select_bsd` function completes successfully and returns 0.

On successful completion, the objects pointed to by the **reads**, **writes**, and **excepts** arguments are modified to indicate which file descriptors are ready. The file descriptors in each file descriptor set input that are not ready are removed from the respective file descriptor set.

If the **reads**, **writes**, and **excepts** arguments are all null pointers and the **tv** argument is not a null pointer, the `tpf_select_bsd` function blocks for the time interval specified. If the **reads**, **writes**, and **excepts** arguments are all null pointers and the **tv** argument is a null pointer, the `tpf_select_bsd` function returns immediately.

On error return, the objects pointed to by the **reads**, **writes**, and **excepts** arguments are not modified.

### Normal Return

The number of ready file descriptors. A value of 0 indicates an expired time limit. If the return value is greater than 0, the **reads**, **writes**, and **excepts** arguments are modified.

## tpf\_select\_bsd

### Error Return

A value of -1 indicates an error. See “select—Monitor Read, Write, and Exception Status” on page 450 for possible sock\_errno values.

### Programming Considerations

- The select function is designed to handle a larger number of file descriptors than the tpf\_select\_bsd function. In order to aid applications that you are porting to the TPF system, the related macros and structures support integer lists of file descriptors rather than Berkeley Software Distribution (BSD) bitmap model. To port an application using the tpf\_select\_bsd function, you must change all occurrences of the select function to the tpf\_select\_bsd function. There are no parameter changes necessary to support large socket descriptors or a large number sockets in the TPF system.
- If you want to monitor more than 256 file descriptors on a single tpf\_select\_bsd call, the application must define its own FD\_SETSIZE before the include of header sys/time.h.

### Examples

The following example shows a simplified server using the BSD style functions to listen on a socket and accept new connections.

```
#include <stdio.h>
#include <stdlib.h>
#include <socket.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    fd_set rfd2, rfd;
    fd_set efd2, efd;
    struct timeval tv;
    int myfds[FD_SETSIZE];
    int rc;
    int newsock;
    int maxsock = 1;
    int i;

    /* establish listener in first position */
    myfds[0] = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (myfds[0] < 0)
        exit(1);
    rc = listen(myfds[0], 5);
    if (rc < 0)
        exit(1);
    FD_ZERO(&rfd2);
    FD_ZERO(&efd2);

    FD_SET(myfds[0], &rfd2); /* add listener socket */
    /* Wait up to 60 seconds. */
    tv.tv_sec = 60;
    tv.tv_usec = 0;
    FD_COPY(&rfd2, &rfd);
    FD_COPY(&efd2, &efd);

    /* loop forever unless error from select */
    while ((rc = tpf_select_bsd(maxsock, &rfd2, NULL, &efd2, &tv)) >= 0) {
        /* when listener is ready, there is a new socket to accept */
        if (FD_ISSET(myfds[0], &rfd2)) {
            newsock = accept(myfds[0], NULL, NULL);
            if (newsock > 0) {
```

```

        FD_SET(newsock, &rfd2);
        FD_SET(newsock, &efd2);
        myfds[maxsock++] = newsock;
        my_accept(&newsock); /* call a function to handle the new connection */
    }
}
my_reads(&myfds, &rfd2); /* call a function to handle reads */
for (i = 0; i < maxsock; i++) /* look for errors */
    if (FD_ISSET(myfds[i], &efd2)) {
        /* close socket, remove from list */
        close(myfds[i]);
        FD_CLR(myfds[i], &rfd2);
        FD_CLR(myfds[i], &efd2);
    }
/* last thing before returning to top of loop, reestablish the file descriptors
   used by select */
FD_COPY(&rfd2, &rfd2);
FD_COPY(&efd2, &efd2);
}
exit(1); /* if we get here, select had an error */
}

```

## Related Information

- “FD\_CLR—Remove File Descriptor from the File Descriptor Set” on page 134
- “FD\_COPY—Copy the File Descriptor Set” on page 135
- “FD\_ISSET—Return a Value for the File Descriptor in the File Descriptor Set” on page 136
- “FD\_SET—Add a File Descriptor to a File Descriptor Set” on page 139
- “FD\_ZERO—Initialize the File Descriptor Set” on page 140
- “select—Monitor Read, Write, and Exception Status” on page 450

See *TPF Transmission Control Protocol/Internet Protocol* for more information about socket file descriptors and the `sock_errno` function.

## tpf\_snmp\_BER\_encode—Encode SNMP Variables in BER Format

This function encodes Simple Network Management Protocol (SNMP) variables using a subset of the basic encoding rules (BER). Use this function when passing variables from the UMIB user exit and when encoding a variable binding list for the tpf\_itrpc C/C++ function or the ITRPC macro. For information about BER encoding, see ISO 8825 *Part 1: Basic Encoding Rules*. Go to <http://www.iso.ch/> to view ISO 8825.

### Format

```
#include <c$snmp.h>
int tpf_snmp_BER_encode(struct snmp_struct *snmp_structure_ptr);
```

#### snmp\_structure\_ptr

A pointer to snmp\_struct. This structure contains the following elements:

#### snmp\_input\_type

The unsigned character defining the type of variable to encode. Use one of the following values:

##### ISNMP\_TYPE\_INTEGER

For integer-type values.

##### ISNMP\_TYPE\_DISPLAYSTRING

For display string-type values.

##### ISNMP\_TYPE\_OBJECTID

For object ID-type values.

##### ISNMP\_TYPE\_IP\_ADDRESS

For Internet Protocol (IP) address-type values.

##### ISNMP\_TYPE\_COUNTER

For counter-type values.

##### ISNMP\_TYPE\_GAUGE

For gauge-type values.

##### ISNMP\_TYPE\_TIMETICKS

For time ticks-type values.

##### ISNMP\_TYPE\_SEQUENCE\_OF

For sequence of-type values.

#### snmp\_input\_length

The integer value defining the length of the variable to encode that is pointed to by **snmp\_input\_value**. Valid lengths are in the range 1–32 767 bytes.

**Note:** When using SNMP agent support, the maximum packet size is 548.

#### snmp\_input\_value

The pointer to the value to be encoded.

#### snmp\_output\_value

The pointer to the storage in which to build the variable encoded in BER format.

### Normal Return

A value of 0. The following field in snmp\_struct was updated:

**snmp\_output\_length**

The integer value defining the length of the encoded value that is pointed to by **snmp\_output\_value**.

**snmp\_output\_value**

The pointer that now contains the BER-encoded value.

## Error Return

A value of -1. An unsupported type or length was passed. No elements in snmp\_struct were updated.

## Programming Considerations

This function is implemented in dynamic link library (DLL) CNMP. You must use the definition side-deck for DLL CNMP to link-edit an application that uses this function.

## Examples

The following example encodes a 7-byte object identifier pointed to by oid\_ptr in BER format.

```
#include <c$snmp.h>
#include <stdlib.h>
#include <string.h>

extern "C" int QZZ8()
{
    struct    snmp_struct snmp_structure_ptr;
    char      buffer_ptr[100];
    char      *oid_ptr;
    int       rc,length=0;

    oid_ptr = (char *)malloc(24);
    memcpy(oid_ptr, "\\x2B\\x06\\x01\\x02\\x01\\x04\\x01", 7);

    snmp_structure_ptr.snmp_input_type    = ISNMP_TYPE_OBJECTID;
    snmp_structure_ptr.snmp_input_length = 7;
    snmp_structure_ptr.snmp_input_value  = oid_ptr;
    snmp_structure_ptr.snmp_output_value = buffer_ptr;

    rc = tpf_snmp_BER_encode(&snmp_structure_ptr);
    if (rc == -1)
        return(-1);

    length += snmp_structure_ptr.snmp_output_length;
    return (0);
}
```

## Related Information

- “tpf\_itrpc—Send Simple Network Management Protocol User Trap” on page 608.
- *TPF Transmission Control Protocol/Internet Protocol* for information about SNMP agent support.

---

## tpf\_STCK–Store Clock

This function returns a pointer to the current value of the time-of-day (TOD) clock.

### Format

```
include    <c$stck.h>
int        tpf_STCK(tpf_TOD_type *tod);
```

#### tod

A pointer of type tpf\_TOD\_type representing the address of the current value of the TOD clock.

### Normal Return

STCK\_OK is returned.

### Error Return

STCK\_ERROR is returned if the clock is in the error state, not-set state, or in the not-operational state.

### Programming Considerations

None.

### Examples

The following example gets the address of the current value of the TOD clock.

```
#include <c$stck.h>

tpf_TOD_type    tod_time;

if(tpf_STCK(&tod_time) != STCK_OK)
{
    /* tpf_STCK error */
}
```

### Related Information

None.



## tpf\_tcpip\_message\_cnt—Update the Message Counters for TCP/IP Applications

This function updates the message counters for TCP/IP applications.

### Format

```
include <socket.h>
void tpf_tcpip_message_cnt(int cnt, int port, const char *proto, int messages);
```

#### cnt

An integer that specifies which counter to update. Specify one of the following:

#### NSDB\_OUTPUT\_CNT

Updates the output message counter.

#### NSDB\_INPUT\_CNT

Updates the input message counter.

#### port

The port of the server application.

#### proto

The protocol of the server application.

#### messages

The number of messages by which to increment the counter.

### Normal Return

Void.

### Error Return

Not Applicable.

### Programming Considerations

- This function is valid only for applications that use TCP/IP native stack support.
- The application must be defined in the TCP/IP network services database.

### Examples

The following example updates the inbound message counter for a TCP/IP server application at port 5001.

```
#include <socket.h>

int cnt;
int port;
int messages;
char proto[4] = "TCP";

port = 5001;
messages = 1;
cnt = NSDB_INPUT_CNT;

tpf_tcpip_message_cnt(cnt, port, proto, messages);
```

### Related Information

See *TPF Transmission Control Protocol/Internet Protocol* for more information about TCP/IP network services database support.

---

## tpf\_tm\_getToken—Get the Unique Token for the Current Transaction

This function gets the unique token for the current transaction.

### Format

```
include    <c$tmcr.h>
double     tpf_tm_getToken(void);
```

### Normal Return

The tpf\_tm\_getToken function returns one of the following:

- A nonzero unique transaction token for the current transaction in which the entry control block (ECB) is running.
- A zero, which indicates that the ECB is not currently in an active transaction.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example gets the unique token for the current transaction.

```
#include <c$tmcr.h>
test_pgm(void)
{
    double token;

    tx_begin();
    tx_suspend_tpf();
    token = tpf_tm_getToken();    /* token should be zero */
                                /* no active transaction */

    tx_resume_tpf();

    token = tpf_tm_getToken();    /* returns the transaction token */
                                /* for the current transaction */

    tx_commit();
    return;
}
```

### Related Information

- “tx\_begin—Begin a Global Transaction” on page 647
- “tx\_resume\_tpf—Resume a Global Transaction” on page 652
- “tx\_suspend\_tpf—Suspend a Global Transaction” on page 655.

## tpf\_vipac—Move a VIPA to Another Processor

This function moves a virtual IP address (VIPA) to another processor in the same loosely coupled complex.

### Format

```
#include <socket.h>
int tpf_vipac(unsigned long *vipa, char *cpu);
```

#### **vipa**

A pointer to a 4-byte virtual IP address.

#### **cpu**

A pointer to the 1-byte TPF processor ID.

### Normal Return

A value of 0. This value indicates that the TPF system was able to start moving the VIPA to the specified processor.

### Error Return

One of the following values, indicating the respective error:

- |           |  |
|-----------|--|
| <b>1</b>  | Incorrect VIPA.                              |
| <b>2</b>  | VIPA not defined as movable.                 |
| <b>3</b>  | Incorrect CPU.                               |
| <b>4</b>  | CPU not active.                              |
| <b>5</b>  | System not in 1052 state or above.           |
| <b>6</b>  | OSA-Express support is not defined.          |
| <b>7</b>  | VIPA not defined on the specified processor. |
| <b>8</b>  | VIPA already moving.                         |
| <b>9</b>  | CPU already owns the VIPA.                   |
| <b>10</b> | Internal system error.                       |

### Programming Considerations

- This function can be run on any I-stream.
- The system must be in 1052 state or above.
- To move a VIPA, you must define it as movable to the processor where you want it moved.

### Examples

The following example attempts to move VIPA 9.155.155.155 (0x99B9B9B) to CPU C.

```
#include <socket.h>

int rc
unsigned long vipa=0x99B9B9B;
char cpu='c';

rc= tpf_vipac(&vipa,&cpu);
```

**tpf\_vipac**

## **Related Information**

See *TPF Migration Guide: Program Update Tapes*, *TPF Operations* , and *TPF Transmission Control Protocol/Internet Protocol* for more information about moving VIPAs.

## tpf\_yieldc—Yield Control

This function gives up control of the processor and allows other entry control blocks (ECBs) to be processed. The ECB is placed on the specified processor list.

### Format

```
#include <sysapi.h>
void tpf_yieldc(enum t_yieldc_opt list);
```

#### list

Specifies the type of list on which you want the ECB placed. Use one of the following values:

#### YIELD ready\_list

Assigns the ECB to the ready list. This is the highest priority list available to which the ECB can be assigned. Assigning the ECB to this list prevents any new work from being processed by this I-stream before the ECB is dispatched again. Any ECBs that are already on the ready list will be allowed to process.

#### YIELD vct\_list

Assigns the ECB to the virtual file access count (VCT) list. This list is used to delay the processing of an ECB until all work on the ready list has been processed. Any ECB on the VCT list is interleaved with new work. ECBs on the VCT list will still be processed, even when the system stops processing new work because a shutdown condition caused by low resources occurred.

### Normal Return

Void.

### Error Return

Not Applicable.

## Programming Considerations

- Limit the use of this function; excessive use will cause a low-core condition in the real-time system.
- This function can be run on any I-stream.
- After this function is completed, the ECB is added to the specified CPU list. Accordingly, control can be transferred to process another ECB.
- Records must not be held by the ECB when the tpf\_yieldc function is issued. If the ECB that is issuing tpf\_yieldc is holding a record, a number of ECBs (also holding storage blocks) can be queued for that record. This can lead to a lockout condition.
- When this function is issued, the 500-millisecond program timeout is reset.
- The macro trace collection for the tpf\_yieldc function compresses multiple occurrences of tpf\_yieldc to two entries in the macro trace table. This prevents programs that are issuing successive tpf\_yieldc calls from filling up the macro trace table by keeping only the first and last tpf\_yieldc trace information. The first tpf\_yieldc entry contains a count of suppressed entries in addition to the normal macro trace information.
- When this function is processed during system cycle down, tpf\_yieldc is handled like the defrc function by the TPF system and the ECB is placed on the defer list.

## tpf\_yieldc

- If an incorrect list type is specified, the result is a system error with exit.

## Examples

The following example adds the ECB to the ready list.

```
#include <sysapi.h>

:

tpf_yieldc(YIELDC_READY_LIST);
```

## Related Information

“defrc—Defer Processing of Current Entry” on page 80.

## tprdc—Read General Tape Record

This function reads a record from an assigned general tape.

### Format

```
#include <tpftape.h>
void      tprdc(const char *name, enum t_lvl level, enum t_blktype size);
```

#### name

This argument is a pointer to type `char`, which must be a 3-character string identifying the tape to be read from. This function can only be called for a general tape.

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The record being read from tape is attached to this level.

#### size

Storage block list type which is placed on the indicated CBRW. This argument must belong to the enumeration type `t_blktype`, defined in `tpfapi.h`. The record length incoming from tape must match the size of the indicated pool storage type. Use `Lx` notation, where `x` is a valid pool type:

**L0** 127 bytes

**L1** 381 bytes

**L2** 1055 bytes

**L4** 4095 bytes.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- The tape specified as argument **name** must have been previously opened.
- The CBRW level specified by argument **level** must be unoccupied at function call time.
- Use this function to cause a record to be read from a general tape at the current position. The `waitc` function must be called to ensure that the operation has completed. The application is responsible for establishing a pointer to the record on completion.
- Prior to calling `tprdc`, the tape must have been assigned to the issuing ECB via `topnc` or `tasnc`.
- If the tape is mounted in blocked mode, the application should not assume that the execution of this function causes either any physical I/O to be initiated or the application timeout value to be reset.

### Examples

The following example reads a 1055-byte record from the VPH tape to level D2. A message is issued if an I/O error occurs.

## tprdc

```
#include <tpftape.h>
...
tprdc("VPH",D2,L2);
if (waitc())
{
    serrc_op(SERRC_EXIT,0x1234,"ERROR READING VPH TAPE",NULL) ;
}
```

## Related Information

- “tape\_read—Read a Record From General Tape” on page 538
- “waitc—Wait For Outstanding I/O Completion” on page 686
- “tasnc—Assign General Tape to ECB” on page 541
- “topnc—Open a General Tape” on page 556
- “trsvc—Reserve General Tape” on page 644.



## trewc–Rewind General Tape and Wait

This function causes an assigned general tape to be rewound. For multi-volume tape sets, an option may be specified to either rewind the current volume or fallback to the first record of the first volume. Control does not return to the caller until the operation is complete.

### Format

```
#include <tpftape.h>
int      trewc(const char *name, int fallback);
```

#### name

This argument is a pointer to type char, which must be a 3-character string identifying the tape to be rewound. This function can only be called for a general tape.

#### fallback

This argument applies when operating a multi-volume tape file. If **FALLBACK** is specified, the previous volume is mounted when load point is reached on the current volume. If **NO\_FALLBACK** is specified, rewinding stops at the load point with no fallback to the previous volume.

### Normal Return

Integer value of zero indicating successful completion.

### Error Return

The nonzero integer value in CE1SUG is returned to the caller.

## Programming Considerations

- This function calls the equivalent of waitc. To ensure consistent results it is recommended that an explicit waitc function be coded before the call to trewc to ensure that errors for other I/O operations are not reflected in the function return value.
- The tape being rewound must be assigned to the issuing ECB.

### Examples

The following example rewinds the current physical volume of the VPH tape.

```
#include <tpftape.h>
:
:
waitc()
trewc("VPH",NO_FALLBACK);
```

### Related Information

- “tape\_cntl–Tape Control” on page 535
- “tasnc–Assign General Tape to ECB” on page 541
- “topnc–Open a General Tape” on page 556
- “trsvc–Reserve General Tape” on page 644.

---

## trsvc—Reserve General Tape

This function releases an assigned general tape so that another ECB may assign it. No more tape I/O requests for the specified tape may be issued by this ECB until the tape has been reassigned using `tasnc`.

### Format

```
#include <tpftape.h>
void    trsvc(const char *name);
```

#### **name**

This argument is a pointer to type `char`, which must be a 3-character string identifying the tape to be reserved. This function can only be called for a general tape.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

Use this function to permit other ECBs to have I/O access to the specified tape, or to permit the issuing ECB to call the `exit` function, leaving the tape open.

### Examples

The following example reserves the VPH tape for use by other ECBs.

```
#include <tpftape.h>
:
:
trsvc("VPH");
```

### Related Information

- “`exit`—Exit an ECB” on page 115
- “`tasnc`—Assign General Tape to ECB” on page 541.

## tsync—Synchronize a Tape

This function forces the synchronization of any general or realtime tape. Synchronization ensures that the physical tape position is the same as the application's detection of the tape position. This eliminates any discrepancies introduced by buffering, either logical (tapes mounted in blocked mode) or physical (tapes mounted on a device operating in buffered mode).

### Format

```
#include <tpftape.h>
void      tsync(const char *name);
```

#### name

This argument is a pointer to type char, which must be a 3-character string identifying the tape to be synchronized.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- If a general tape is specified, the tape must be open and assigned. If a real-time tape is specified, it must be online and mounted.
- To ensure completion of the operation, waitc must be called. A waitc call is also recommended prior to the tsync call to clear any outstanding DASD I/O.
- If any hardware errors (or end of volume) occur during the buffer flush operation, an automatic switch to the next volume occurs.
- Realtime tape names can be given aliases, or tape pseudonames. These pseudonames are defined using RTMAP definitions in CEFZ. This mechanism provides a means for mapping many names to just a few tape devices.

## Examples

The following example synchronizes the VPH tape.

```
#include <tpftape.h>
:
:
waitc();
tsync("VPH");
if (waitc())
{
    serrc_op(SERRC_EXIT,0x1234,"ERROR SYNCHRONIZING VPH TAPE",NULL) ;
}
```

## Related Information

- “waitc—Wait For Outstanding I/O Completion” on page 686
- “tasnc—Assign General Tape to ECB” on page 541
- “tape\_close—Close a General Tape” on page 534
- “tdspc—Display Tape Status” on page 545.

---

## twrtc—Write a Record to General Tape

This function writes the working storage block on the specified data level to the specified general tape. The block is then returned to the available pool.

### Format

```
#include <tpftape.h>
void twrtc(const char *name, enum t_lvl level);
```

#### name

This argument is a pointer to type `char`, which must be a 3-character string that identifies the tape to which you write a record. This function can only be called for a general tape.

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). The working storage block on this level is the record to be written to tape.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- Use this function to write a record to a general tape. The specified general tape must be opened and assigned to the current ECB.
- The status of the operation can never be determined by the operational program.

## Examples

The following example writes the working storage block on level D3 to the VPH tape. On return, the block is no longer available to the operational program.

```
#include <tpftape.h>
:
:
tasnc("VPH");
twrtc("VPH",D3);
trsvc("VPH");
```

## Related Information

- “`tasnc`—Assign General Tape to ECB” on page 541
- “`topnc`—Open a General Tape” on page 556
- “`tourc`—Write Real-Time Tape Record/Release Storage Block” on page 557
- “`toutc`—Write Real-Time Tape Record” on page 558.

## tx\_begin—Begin a Global Transaction

This function places the calling entry control block (ECB) in transaction mode. The calling ECB must first ensure that the resource managers have been opened (using the `tx_open()` function) before it can start transactions.

Once in transaction mode, the calling ECB must call the `tx_commit` or `tx_rollback` function to complete its current transaction.

The TX specification published by X/Open Company Limited does not allow the `tx_begin()` function to be issued if the caller is already in transaction mode. However, the TPF implementation will allow the following calling sequences:

- If a `tx_begin()` function is issued while the caller is already in transaction mode, the TPF system will begin a nested transaction in the root transaction.
- All subsequent work done by resource managers will become part of the nested transaction until the `tx_commit()` or `tx_rollback()` function is issued.
- Once the nested transaction is ended by the `tx_commit` or `tx_rollback` function, the root transaction becomes the active transaction.
- All work that is done in the nested transaction will become part of its root transaction when the `tx_commit` function is issued for the nested transaction.

## Format

```
#include <tpfapi.h>
int      tx_begin();
```

## Normal Return

Table 22. *tx\_begin* Normal Return

Value Name	Return Code	Description
TX_OK	0	The function is completed successfully.

## Error Return

Table 23. *tx\_begin* Error Return

Value Name	Return Code	Description
TX_OUTSIDE	–1	Resource managers are not opened.

## Programming Considerations

- Resource managers that are XA-compliant must be successfully opened to be included in the global transaction.

## Examples

The following example shows how the `tx_begin` function is used to begin a root transaction and then a nested transaction.

```
#include <tpfapi.h>
      tx_begin();                /* begin a root transaction    */
      .                          /* all RM native API calls will */
      .                          /* be part of the root         */
      .                          /* transaction                  */
      .
      tx_begin();                /* begin a nested transaction  */
```

## tx\_begin

```

    .
    .
    .
    .
    .
    tx_commit();
    .
    .
    .
    .
    tx_commit();

```

```

/* all RM native API calls will */
/* be part of the nested */
/* transaction */
/* commit the nested transaction */
/* all RM native API calls issued*/
/* from the nested transaction */
/* are now rolled in to the root */
/* transaction */
/* commit the root transaction */

```

## Related Information

- “tx\_open—Open a Set of Resource Managers” on page 651
- “tx\_commit—Commit a Global Transaction” on page 649
- “tx\_rollback—Roll Back a Global Transaction” on page 654
- “tx\_suspend\_tpf—Suspend a Global Transaction” on page 655
- “tx\_resume\_tpf—Resume a Global Transaction” on page 652.

## tx\_commit–Commit a Global Transaction

This function commits the work of the transaction that is activated by the entry control block (ECB) of a caller.

### Format

```
#include <tpfapi.h>
int tx_commit();
```

### Normal Return

Table 24. *tx\_commit* Normal Return

Value Name	Return Code	Description
TX_OK	0	The function is completed successfully.

### Error Return

Table 25. *tx\_commit* Error Return

Value Name	Return Code	Description
TX_PROTOCOL_ERROR	-5	The function was called incorrectly.

## Programming Considerations

- TX\_PROTOCOL\_ERROR will be returned if the ECB is not in transaction mode when the tx\_commit function is issued.
- If the tx\_commit function is issued from a nested transaction, the work of the transaction will be rolled into the next higher level transaction.
- If the tx\_commit is issued from a root transaction, the work of the transaction will be committed.

## Examples

The following example shows how the tx\_commit function is used to commit a nested transaction and then the root transaction.

```
#include <tpfapi.h>
tx_begin();                                /* begin a root transaction */
.
.                                           /* all RM native API calls will */
.                                           /* be associated with the root */
.                                           /* transaction */
.
tx_begin();                                /* begin a nested transaction */
.
.                                           /* all RM native API calls will */
.                                           /* be associated with the nested */
.                                           /* transaction */
.
tx_commit();                               /* commit the nested transaction */
.                                           /* all RM native API calls issued */
.                                           /* from the nested transaction is */
.                                           /* now associated with the root */
.                                           /* transaction */
.
tx_commit();                               /* commit the root transaction */
```

**tx\_commit**

## **Related Information**

- “tx\_open—Open a Set of Resource Managers” on page 651
- “tx\_rollback—Roll Back a Global Transaction” on page 654
- “tx\_suspend\_tpf—Suspend a Global Transaction” on page 655
- “tx\_resume\_tpf—Resume a Global Transaction” on page 652.



## tx\_open–Open a Set of Resource Managers

This function opens a set of resource managers. The TPF restart program calls the tx\_open function to open all registered resource managers at restart time.

### Format

```
#include <tpfapi.h>
int tx_open();
```

### Normal Return

Table 26. tx\_open Normal Return

Value Name	Return Code	Description
TX_OK	1	The function is completed successfully.

### Error Return

Table 27. tx\_open Error Return

Value Name	Return Code	Description
TX_FATAL	-7	Resource manager error.

## Programming Considerations

This function can only be called from the TPF restart program.

**Note:** The TPF system does not support a tx\_close function. All TPF resource managers remain open after transaction manager restart is completed.

### Examples

The following example shows a tx\_open call to open a set of resource managers.

```
#include <tpfapi.h>
tx_open();
```

### Related Information

“tx\_begin–Begin a Global Transaction” on page 647.

## tx\_resume\_tpf–Resume a Global Transaction

This function is a TPF-defined extension to the TX specification published by X/Open Company Limited.

This function resumes a transaction that the entry control block (ECB) of the caller suspended.

### Format

```
#include <tpfapi.h>
int      tx_resume_tpf();
```

### Normal Return

Table 28. tx\_resume\_tpf Normal Return

Value Name	Return Code	Description
TX_OK	0	The function is completed successfully.

### Error Return

Table 29. tx\_resume\_tpf Error Return

Value Name	Return Code	Description
TX_PROTOCOL_ERROR	–5	The function was called incorrectly.
TX_FATAL	–7	The transaction is not suspended.

## Programming Considerations

- TX\_PROTOCOL\_ERROR will be returned if the ECB is not in transaction mode when the tx\_resume\_tpf function is called.
- The TPF system allows the tx\_suspend\_tpf function to be issued while the transaction is already suspended. Each suspend will increment a suspend counter associated with the transaction by 1.
- A tx\_resume\_tpf function will decrement the suspend counter associated with the transaction by 1. The transaction is resumed only when the counter reaches zero.

## Examples

The following example shows some of the ways that the tx\_suspend\_tpf and tx\_resume\_tpf functions can be used in either root transaction or nested transaction scenarios. The comments to the right of the example code describe which transactions are active and what value is contained in the suspend counter throughout the example.

Transaction 1 is a root transaction that is suspended with the tx\_suspend\_tpf function before beginning transaction 2 (another root transaction). Transaction 2 is then suspended twice and resumed twice. After the second resume, the suspend counter is zero, so transaction 2 becomes active again. Transaction 2 continues processing and eventually commits. Transaction 1 remains suspended until it is explicitly resumed by the next tx\_resume\_tpf call.

Next, transaction 3 begins as a nested transaction under transaction 1. The tx\_suspend\_tpf call then suspends both transaction 3 and transaction 1.

**Note:** At this point, no transaction is active and any records filed here would be outside of commit scope protection.

Later, the transactions are both resumed with the tx\_resume\_tpf function. Eventually the nested transaction commits and the root transaction also commits.

```
#include <tpfapi.h>
tx_begin();           /* begin transaction 1, root      */
.
tx_suspend_tpf();     /* suspend transaction 1,      */
.                     /* suspend counter = 1.       */
.
tx_begin();           /* begin transaction 2, root   */
.                     /* not a nested transaction   */
.                     /* transaction 2 is now active */
.
tx_suspend_tpf();     /* suspend transaction 2      */
.                     /* suspend counter = 1.       */
.
tx_suspend_tpf();     /* suspend transaction 2 again */
.                     /* suspend counter = 2.       */
.
tx_resume_tpf();      /* resume transaction 2        */
.                     /* suspend counter = 1.       */
.                     /* transaction is still suspended*/
.
tx_resume_tpf();      /* resume transaction2        */
.                     /* suspend counter = 0.       */
.                     /* transaction 2 is now active */
.
tx_commit();          /* commit transaction 2        */
.
.                     /* transaction 1 is suspended */
.                     /* suspend counter is 1.       */
.
tx_resume();          /* transaction 1 is now active */
.                     /* suspend counter is 0.       */
.
.
tx_begin();           /* begin transaction 3, nested */
.
.
tx_suspend_tpf();     /* suspend transaction 3 and 1 */
.                     /* suspend counter is 1.       */
.
tx_resume_tpf();      /* resume transaction 3 and 1   */
.                     /* suspend counter is 0.       */
.                     /* transaction 3 is now active */
.
tx_commit();          /* commit transaction 3        */
.                     /* transaction 1 is now active */
.
tx_commit();          /* commit transaction 1
```

## Related Information

- “tx\_open—Open a Set of Resource Managers” on page 651
- “tx\_commit—Commit a Global Transaction” on page 649
- “tx\_rollback—Roll Back a Global Transaction” on page 654
- “tx\_suspend\_tpf—Suspend a Global Transaction” on page 655.

## tx\_rollback—Roll Back a Global Transaction

This function rolls back the work of the transaction activated by the entry control block (ECB) of the caller.

### Format

```
#include <tpfapi.h>
int      tx_rollback();
```

### Normal Return

Table 30. tx\_rollback Normal Return

Value Name	Return Code	Description
TX_OK	0	The function is completed successfully.

### Error Return

Table 31. tx\_rollback Error Return

Value Name	Return Code	Description
TX_PROTOCOL_ERROR	-5	The function was called incorrectly.

## Programming Considerations

- TX\_PROTOCOL\_ERROR will be returned if the ECB is not in transaction mode when the tx\_rollback function is called.

## Examples

The following example shows how the tx\_rollback function is used to roll back a nested transaction and then the root transaction.

```
#include <tpfapi.h>
tx_begin();           /* begin a root transaction */
.
.
.
.
tx_begin();           /* begin a nested transaction */
.
.
.
.
tx_rollback();         /* rollback nested transaction */
.
tx_rollback();         /* rollback root transaction */
```

## Related Information

- “tx\_open—Open a Set of Resource Managers” on page 651
- “tx\_commit—Commit a Global Transaction” on page 649
- “tx\_suspend\_tpf—Suspend a Global Transaction” on page 655
- “tx\_resume\_tpf—Resume a Global Transaction” on page 652.

## tx\_suspend\_tpf—Suspend a Global Transaction

This function is a TPF-defined extension to the TX specification that is published by X/Open Company Limited.

This function is used to suspend the transaction that the calling entry control block (ECB) is in. Any work that is done by the ECB of the caller will not be considered part of the suspended transaction until a tx\_resume\_tpf function is issued.

### Format

```
#include <tpfapi.h>
int tx_suspend_tpf();
```

### Normal Return

Table 32. tx\_resume\_tpf Normal Return

Value Name	Return Code	Description
TX_OK	0	The function is completed successfully.

### Error Return

Table 33. tx\_resume\_tpf Error Return

Value Name	Return Code	Description
TX_PROTOCOL_ERROR	-5	The function was called incorrectly.

## Programming Considerations

- TX\_PROTOCOL\_ERROR is returned if the ECB is not in transaction mode when the tx\_suspend\_tpf function is called.
- The TPF system allows the tx\_suspend\_tpf function to be issued while the transaction is already suspended. Each suspend request will increment a suspend counter associated with the transaction by 1.
- A tx\_resume\_tpf function will decrement the suspend counter associated with the transaction by one. The transaction is resumed only when the counter reaches zero.
- An independent transaction will be created if a tx\_begin function is issued while the current transaction is suspended.

## Examples

The following example shows some of the ways that the tx\_suspend\_tpf and tx\_resume\_tpf functions can be used in either root transaction or nested transaction scenarios. The comments to the right of the example code describe which transactions are active and what value is contained in the suspend counter throughout the example.

Transaction 1 is a root transaction that is suspended using the tx\_suspend\_tpf function before beginning transaction 2 (another root transaction). Transaction 2 is then suspended twice and resumed twice. After the second resume, the suspend counter is zero, so transaction 2 becomes active again. Transaction 2 continues processing and eventually commits. Transaction 1 remains suspended until it is explicitly resumed by the next tx\_resume\_tpf call.

## tx\_suspend\_tpf

Next, transaction 3 begins as a nested transaction under transaction 1. The tx\_suspend\_tpf call then suspends both transaction 3 and transaction 1.

**Note:** At this point, no transaction is active and any records filed here would be outside of commit scope protection. Later, both transactions are resumed by the tx\_resume\_tpf function. Eventually the nested transaction commits and the root transaction also commits.

```
#include <tpfapi.h>
tx_begin();           /* begin transaction 1, root    */
.
tx_suspend_tpf();     /* suspend transaction 1,      */
.                   /* suspend counter = 1.        */
.
tx_begin();           /* begin transaction 2, root    */
.                   /* not a nested transaction    */
.                   /* transaction 2 is now active */
.
tx_suspend_tpf();     /* suspend transaction 2       */
.                   /* suspend counter = 1.        */
.
tx_suspend_tpf();     /* suspend transaction 2 again  */
.                   /* suspend counter = 2.        */
.
tx_resume_tpf();      /* resume transaction 2         */
.                   /* suspend counter = 1.        */
.                   /* transaction is still suspended*/
.
tx_resume_tpf();      /* resume transaction2         */
.                   /* suspend counter = 0.        */
.                   /* transaction 2 is now active */
.
tx_commit();          /* commit transaction 2         */
.
.                   /* transaction 1 is suspended  */
.                   /* suspend counter is 1.        */
.
tx_resume();          /* transaction 1 is now active  */
.                   /* suspend counter is 0.        */
.
.
tx_begin();           /* begin transaction 3, nested */
.
.
tx_suspend_tpf();     /* suspend transaction 3 and 1  */
.                   /* suspend counter is 1.        */
.
.
tx_resume_tpf();      /* resume transaction 3 and 1   */
.                   /* suspend counter is 0.        */
.                   /* transaction 3 is now active */
.
tx_commit();          /* commit transaction 3         */
.                   /* transaction 1 is now active */
.
.
tx_commit();          /* commit transaction 1
```

## Related Information

- “tx\_open—Open a Set of Resource Managers” on page 651
- “tx\_commit—Commit a Global Transaction” on page 649
- “tx\_rollback—Roll Back a Global Transaction” on page 654
- “tx\_resume\_tpf—Resume a Global Transaction” on page 652.

## uatbc–MDBF User Attribute Reference Request

This function is used to provide addressability to an SSU slot in the Subsystem User Table (MSOUT). The function calculates the requested SSU slot address and returns a pointer to the slot in the **uatbc\_area**. The user can access the data in the SSUT via the msOut structure (see header file c\$msOut).

### Format

```
#include <c$uatbc.h>
int      *uatbc(struct uatbc_area *uatbc_area_ptr);
```

#### uatbc\_area\_ptr

A pointer to a work area for uatbc. This area is used for passing parameters to uatbc and for holding return information.

#### uatbc\_area\_ptr-> uat\_call

Following are the possible values for uat\_call. Each value indicates to uatbc what and where is the input parameter.

##### UAT\_ESSI

Input is the SS ID in the ECB, ce1dbi

##### UAT\_ESSU

Input is the SSU ID in the ECB, ce1ssu

##### UAT\_RSSI

Input is the SS ordinal number in the **uatbc\_area**, **uat\_parm**. The SS ordinal number must be in the left-most byte of **uat\_parm**.

##### UAT\_RSSU

Input is the SSU ordinal number in the **uatbc\_area**, **uat\_parm**. The SS ordinal number must be in the left-most byte of **uat\_parm**.

##### UAT\_NSSI

Input is the SS name in the **uatbc\_area**, **uat\_parm**.

##### UAT\_NSSU

Input is the SSU name in the **uatbc\_area**, **uat\_parm**.

#### uatbc\_area\_ptr->uat\_parm

The area contains the addition parameter as specified by **uat\_call**. The contents of **uat\_parm** must be left-justified.

### Normal Return

#define UAT\_SUCCESS Return code 0.

### Error Return

Table 34. uatbc Error Return

Value Name	Return Code	Description
#define UAT_EXCD	8	Either the SS ordinal number exceeds the number of subsystems included in the last IPL or the SSU ordinal exceeds the number of subsystem users in the parent subsystem.
#define UAT_INVALID	16	Either the SS ID in CE1DBI or the SSU ID in CE1SSU is not valid.
#define UAT_NOTAVL	32	Either the specified SS is inactive or the specified SSU is dormant.

## Programming Considerations

The `msOut` structure is used to map the SSU entries that are returned.

### `uatbc_area_ptr->uat_ssu_ptr`

Contains the pointer to an SSU slot. `struct msOut` should be used to map this slot.

- If `uat_call` was **UAT\_xSSI**, then the SSU slot returned is that of the first SSU within the specified SS.
- If `uat_call` was **UAT\_xSSU**, then the SSU slot returned is that of the specified SSU.

### `uatbc_area_ptr->uat_ss_name`

Contains the SS name, left justified, of the parent SS to the SSU referenced by `uat_ssu_ptr`.

### `uatbc_area_ptr->uat_ssu_cnt`

Contains a count of SSU entries.

- If `uat_call` was **UAT\_xSSI**, then the count is the number of SSU entries within the specified SS.
- If `uat_call` was **UAT\_xSSU**, then the count is the number of remaining SSU entries within the parent SS including the specified SSU. For example, if there are 5 SSUs in the SS and the specified SSU is the second one, then the count would be 4.

## Examples

The following example uses the SSU ID in the ECB, retrieves the SSU entry, and copies the SS name and SSU name to an area specified by `tmp_var_ptr`.

```
#include <c$msOut.h>
#include <c$uatbc.h>
#include <string.h>
:
:
char * tmp_var_ptr;
struct uatbc_area uatbc_parm;
struct msOut *ssu_slot_ptr;
:
:
uatbc_parm.uat_call = UAT_ESSU;
if (uatbc(&uatbc_parm) == 0)
{
    ssu_slot_ptr = uatbc_parm.uat_ssu_ptr;
    (void) memcpy(tmp_var_ptr,
                  &uatbc_parm.uat_ss_name,
                  sizeof(uatbc_parm.uat_ss_name));
    tmp_var_ptr += sizeof(uatbc_parm.uat_ss_name);
    (void) memcpy(tmp_var_ptr,
                  ssu_slot_ptr->mu0nam,
                  sizeof(ssu_slot_ptr->mu0nam));
    tmp_var_ptr += sizeof(ssu_slot_ptr->mu0nam);
}
:
:
```

## Related Information

- “`cebic_save`—Save the Current DBI and SSU IDs” on page 29
- “`cebic_restore`—Restore Previously Saved DBI and SSU IDs” on page 28
- “`cebic_goto_bss`—Change MDBF Subsystem to BSS” on page 25
- “`cebic_goto_dbi`—Change MDBF Subsystem to DBI” on page 26
- “`cebic_goto_ssu`—Change MDBF Subsystem” on page 27.



## umask—Set the File Mode Creation Mask

This function sets the file mode creation mask.

### Format

```
#include <sys/stat.h>
mode_t umask(mode_t newmask);
```

#### **newmask**

Specifies the new file permission bits for the file creation mask of the process.

This function changes the file creation mask of the process.

This mask controls file permission bits that should be set whenever the process creates a file. File permission bits that are set to 1 in the file creation mask are set to 0 in the file permission bits of files that are created by the process.

For example, if a call to the open function specifies a **mode** argument with file permission bits, the file creation mask of the process affects the **mode** argument; bits that are 1 in the mask are set to 0 in the **mode** argument and, therefore, in the mode of the created file.

Only the file permission bits of the new mask are used. For more information about these symbols, see to “chmod—Change the Mode of a File or Directory” on page 32.

### Normal Return

The umask function is always successful and returns the previous value of the file creation mask.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

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

main()
{
    int fd;
    int save_mask = umask(S_IRWXG);

    if ((fd = creat("umask.file", S_IRWXU|S_IRWXG)) < 0)
        perror("creat() error");
    else
    {
        close(fd);
        unlink("umask.file");
    }
}
```

**umask**

## **Related Information**

- “chmod—Change the Mode of a File or Directory” on page 32
- “creat—Create a New File or Rewrite an Existing File” on page 54
- “mkdir—Make a Directory” on page 326
- “open—Open a File” on page 380.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## unfrc—Unhold a File Record

This function removes a record from the record hold table.

### Format

```
#include <tpfio.h>
void unfrc(enum t_lvl level);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This argument identifies the FARW containing the file address of the record to be removed from the record hold table.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The record address residing on the FARW specified by argument **level** must have been placed in the record hold table by the calling ECB. Control is transferred to the system error routine (with exit) if the address is not in the RHT, or the record hold table refers to a different ECB.
- When the file address returns from the `unfrc` call, it appears to be unheld from the point of view of the program. For requests from outside the commit scope, the file address still appears to be held. When the commit is completed successfully, the file address will be unheld and any waiting requests will be serviced.

## Examples

The following example finds (with hold) an application message block on level 6 and then removes the address from the record hold table.

```
#include <tpfio.h>
#include <c$am0sg.h>

struct am0sg *prime, *chain;          /* Pointers to message blocks */
:
prime = ecbptr()->celcr1;             /* Base prime message block */
/* and read 1st chain w/hold */
chain = find_record(D6,(unsigned int *)&(prime->am0fch),"OM","\0",HOLD);
unfrc(D6);                            /* Now remove chain address from
                                     the record hold table. */
```

## Related Information

- “`unfrc_ext`—Unhold a File Record with Extended Options” on page 662
- “`find_record`—Find a Record” on page 178
- “`finhc`—Find and Hold a File Record” on page 185
- “`fiwhc`—Find and Hold a File Record and Wait” on page 193.

## unfrc\_ext—Unhold a File Record with Extended Options

This function removes a record from the record hold table (RHT).

### Format

```
#include <tpfio.h>
void unfrc_ext(enum t_lvl level, unsigned int ext);
```

or

```
#include <tpfio.h>
void unfrc_ext(TPF_DECB *decb, unsigned int ext);
```

#### level

One of 16 possible values representing a valid entry control block (ECB) data level from enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the ECB data level (0–F). This argument identifies the FARW containing the file address of the record to be removed from the RHT.

#### decb

A pointer to a data event control block (DECB). This argument identifies the FARW containing the file address of the record to be removed from the RHT.

#### ext

Sum of the following bit flags, which are defined in `tpfio.h`.

##### FIND\_GDS

Use `FIND_GDS` to specify that the record to be removed from the record hold table resides on a general file or general data set. If `FIND_GDS` is not specified, `unfrc_ext` accesses the online database.

**Note:** If the flag is not needed, the default extended options flag (`FIND_DEFEXT`) should be coded. In this case, consider using the `unfrc` function.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- The record address residing on the FARW specified by the **level** or **decb** parameter must have been placed in the RHT by the calling ECB. Control is transferred to the system error routine (with `exit`) if the address is not in the RHT or the RHT references a different ECB.
- When the file address returns from the `unfrc_ext` call, it appears to be unheld from the point of view of the program. For requests from outside the commit scope, the file address still appears to be held. When the commit is completed successfully, the file address will be unheld and any waiting requests will be serviced.
- Applications that call the function using DECBs instead of ECB data levels must be compiled with the C++ compiler because this function has been overloaded.
- This function is implemented in dynamic link library (DLL) CTAD. You must use the definition side-deck for DLL CTAD to link-edit an application that uses this function.

## Examples

The following example removes the address of the general data set record on level D7 from the record hold table.

```
#include <tpfio.h>
...
unfrc_ext(D7,FIND_GDS);
```

The following example removes the address of the general data set record on a DECB from the record hold table.

```
#include <tpfio.h>
...
TPF_DECB *decb;
...
unfrc_ext(decb,FIND_GDS);
```

## Related Information

- “find\_record\_ext–Find a Record with Extended Options” on page 181
- “finhc\_ext–Find and Hold a File Record with Extended Options” on page 187
- “fiwhc\_ext–Find and Hold a File Record and Wait with Extended Options” on page 195
- “unfrc–Unhold a File Record” on page 661.

## ungetc–Push Character to Input Stream

This function pushes a character to input stream.

### Format

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

**c** A character.

**stream**

The input stream.

This function pushes the character specified by the value of **c** converted to an unsigned char to the given input stream. The pushed back characters are returned by any subsequent read on the same stream in reverse order; that is, the last character pushed back will be returned first.

You can push back as many as 4 characters to a given input stream. You can call the ungetc function as many as four times consecutively; this will result in a total of 4 characters being pushed.

The stream must be open for reading. A subsequent read operation on the stream starts with **c**. You cannot push back end-of-file (EOF) on the stream using the ungetc function. A successful call to the ungetc function clears the EOF indicator for the stream.

Characters pushed back by the ungetc function, and subsequently not read in, will be erased if an fseek, fsetpos, rewind, or fflush function is called before the character is read from the stream. After all the pushed back characters are read in, the file position indicator is the same as it was before the characters were pushed back.

Each pushed back character backs up the file position by 1 byte. This affects the value returned by the ftell or fgetpos functions, the result of an fseek function using SEEK\_CUR, or the result of an fflush function. For example, consider a file that contains a b c d e f g h:

After you have just read 'a', the current file position is at the start of 'b'. The following operations will all result in the file position being at the start of 'a', ready to read 'a' again.

```
/* 1 */      ungetc('a', fp);
              fflush(fp); /* flushes ungetc char and keeps position */

/* 2 */      ungetc('a', fp);
              pos = ftell(fp); /* points to first character */
              fseek(fp, pos, SEEK_SET);

/* 3 */      ungetc('a', fp);
              fseek(fp, 0, SEEK_CUR) /* starts at new file pos'n */

/* 4 */      ungetc('a', fp);
              fgetpos(fp, &fpos); /* gets position of first char */
              fsetpos(fp, &fpos);
```

The ungetc function has the same restriction as any read operation for a read immediately following a write, or a write immediately following a read. Between a

write and a subsequent ungetc function, there must be an intervening flush or reposition. Between an ungetc function and a subsequent write, there must be an intervening reposition.

## Normal Return

Returns integer argument **c** converted to an unsigned char.

## Error Return

If **c** cannot be pushed back, the ungetc function returns EOF.

## Programming Considerations

None.

## Examples

In the following example, the while statement reads decimal digits from an input data stream by using arithmetic statements to compose the numeric values of the numbers as it reads them. When a nondigit character appears before EOF, the ungetc function replaces the nondigit character in the input stream so that later input functions can process it.

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

int main(void)
{
    FILE *stream;
    int ch;
    unsigned int result = 0;

    stream = fopen("myfile.dat", "r+");
    while ((ch = getc(stream)) != EOF && isdigit(ch))
    {
        result = result * 10 + ch - '0';
    }
    printf("result is %i\n", result);
    if (ch != EOF)
    {
        ungetc(ch, stream);          /* Put the nondigit character back */
        ch = getc(stream);
        printf("value put back was %c\n", ch);
    }
}
```

## Related Information

- “fflush—Write Buffer to File” on page 144
- “fseek—Change File Position” on page 226
- “getc, getchar—Read a Character from Input Stream” on page 246
- “putc, putchar—Write a Character” on page 400.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## unhka–Unhook Core Block

The function transfers the information from a requested CBRW to a specified 8-byte field and detaches the storage block associated with the CBRW. If the field specified is in global storage, the global function is used to update the field. Once a block has been *unhooked*, any other ECB can *rehook* the same block by using this function.

### Format

```
#include <tpfapi.h>
void unhka(enum t_lvl level, enum t_hook_type glob_indicator, ...);
```

#### level

One of 16 possible values representing a valid data level from the enumeration type `t_lvl`, expressed as `Dx`, where `x` represents the hexadecimal number of the level (0–F). This argument represents the CBRW from where information is to be transferred from.

#### glob\_indicator

This argument must belong to the enumeration type `t_hook_type` specifying one of two possible values:

##### UNHKA\_GLOBAL

Specifies that the save area is in protected storage.

##### UNHKA\_UNPROTECTED

Specifies that the save area is not in protected storage.

... This argument is either an address if the 8-byte save area is not in protected storage or a global tag if the field is in protected storage.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

- The location of the last parameter will contain the contents of the CBRW specified by `CE1CR(literal)`.
- The use of this function must be limited to those functions having unique buffering requirements. It should not be used to obtain an additional data level.
- Any application function using the `unhka-rehka` facility must be designed so that 2 successive `unhka` functions referencing a given field are always separated by a `rehka` function referencing the field.
- `unhka` may use the global function.
- If the save area supplied is a synchronizable global field that needs to be locked before modified, then it becomes the users responsibility to call the global function with the `GLOBAL_LOCK`, `GLOBAL_UNLOCK` or `GLOBAL_SYNC` option, appropriately.
- If an incorrect parameter is given, a system error with exit occurs.

## Examples

The following example unhooks a core block from level 0 to a location pointed to by `save_ptr`.



```
#include <tpfapi.h>
:
char    save_area[8];
:
unhka(D0, UNHKA_UNPROTECTED, save_area);
:
```

The following example unhooks a core block from level 0 to a location specified by the global tag `_s3hok`.

```
#include <tpfapi.h>
:
unhka(D0, UNHKA_PROTECTED, _s3hok);
:
```

## Related Information

“rehka—Rehook Core Block” on page 419.

## unlink–Remove a Directory Entry

This function removes a directory entry.

### Format

```
#include <unistd.h>
int unlink(const char *pathname);
```

#### **pathname**

The link to be deleted.

The `unlink` function deletes the link named by **pathname** and decrements the link count for the file itself.

**pathname** can refer to a path name, a link, or a symbolic link. If **pathname** refers to a symbolic link, the `unlink` function removes the symbolic link, but not any file or directory named by the contents of the symbolic link.

If any processes have the file open when the `unlink` function removes the last link and the directory entry, the file itself remains available to these processes until they close it. The last process to be completed closes the file.

The `unlink` function cannot be used to remove a directory; use the `rmdir` function instead.

#### TPF deviation from POSIX

The TPF system does not support the `ctime` (change time) time stamp.

### Normal Return

If successful, the `unlink` function updates the modification time for the parent directory and returns a value of 0.

### Error Return

If unsuccessful, the `unlink` function returns a value of `-1` and sets `errno` to one of the following:

- |                     |  |
|---------------------|--|
| <b>EACCES</b>       | The process did not have search permission for some component of <b>pathname</b> , or did not have write permission for the directory containing the link to be removed.   |
| <b>EBUSY</b>        | The link specified by <b>pathname</b> cannot be unlinked because it is currently being used by the system or some other process.   |
| <b>ELOOP</b>        | A loop exists in symbolic links. This error is issued if the number of symbolic links found while resolving the <b>pathname</b> argument is greater than <code>POSIX_SYMLINK_MAX</code> .  |
| <b>ENAMETOOLONG</b> | <b>pathname</b> is longer than <code>PATH_MAX</code> characters or some component of <b>pathname</b> is longer than <code>NAME_MAX</code> characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds <code>PATH_MAX</code> . |
| <b>ENOENT</b>       | <b>pathname</b> does not exist or it is an empty string.   |
| <b>ENOTDIR</b>      | Some component of the <b>pathname</b> prefix is not a directory.   |

**EPERM** **pathname** is a directory and the unlink function cannot be used on directories.

#### ETPFNPIPWSYS

In a loosely coupled environment, there was an attempt to access a FIFO special file (or named pipe) from a processor other than the processor on which the file was created.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

## Examples

The following example removes a directory entry.

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

main() {
    int fd;
    char fn[]="unlink.file";

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        if (unlink(fn) != 0)
            perror("unlink() error");
    }
}
```

## Related Information

- “mkfifo—Make a FIFO Special File” on page 328
- “remove—Delete a File” on page 427
- “close—Close a File” on page 44
- “link—Create a Link to a File” on page 302
- “open—Open a File” on page 380
- “rmdir—Remove a Directory” on page 439.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## unlkc—Unlock a Resource

This function is used to unlock a resource previously locked by the `lockc` function. If the lock is not held by this I-stream, a dump is taken and the lock is unlocked.

### Format

```
#include <tpfapi.h>
int      unlkc(void *lockword);
```

#### **lockword**

A pointer to a *lock* made up of 2 consecutive fullwords used for lock and trace.

### Normal Return

0.

### Error Return

Not applicable.

### Programming Considerations

The lock specified by `lockword` must be held by this I-stream. If the lock is not held, a CTL-000573 will be taken and the lock unlocked.

### Examples

The following example unlocks an area whose lock field is pointed to by `table_lock_ptr`.

```
#include <tpfapi.h>
:
:
unlkc(table_lock_ptr);
:
```

### Related Information

"lockc—Lock a Resource" on page 305.

## unsetenv—Delete an Environment Variable

This function deletes environment variables.

### Format

```
#define _POSIX_SOURCE
#include <stdlib.h>
int unsetenv(const char *var_name);
```

#### **var\_name**

A pointer to a character string that contains the name of the environment variable to be deleted.

### Normal Return

0.

### Error Return

−1 and `errno` is set to `EINVAL`.

## Programming Considerations

- Only the private environment list of the process is changed; `unsetenv` does not change the global default environment list.
- If you call the `unsetenv` function with **var\_name** containing an equal sign (=), the `unsetenv` function fails and `errno` is not set. This indicates that an incorrect argument was passed to the function.

## Examples

The following examples set up and print environment variable `_EDC_ANSI_OPEN_DEFAULT`.

```
#include <stdio.h>
#define _POSIX_SOURCE
#include <stdlib.h>

int main(void)
{
    char *x;

    /* set environment variable _EDC_ANSI_OPEN_DEFAULT to "Y" */
    setenv("_EDC_ANSI_OPEN_DEFAULT", "Y", 1);

    /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT */
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("program1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");

    /* call the child program */
    system("program2");

    /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT */
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("program1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");
}
```

### Output

## unsetenv

```
program1 _EDC_ANSI_OPEN_DEFAULT = Y
program1 _EDC_ANSI_OPEN_DEFAULT = Y
```

**Y** The value defined for the environment variable.

The following example, which is a child of the previous example, is started by a system call. Like the previous example, this example sets up and prints the value of environment variable `_EDC_ANSI_OPEN_DEFAULT` or, as in this case, indicates that the value is undefined. This example shows that environment variables are propagated forward to a child program, but not backward to the parent.

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

int main(void)
{
    char *x;

    /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT*/
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("program2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");

    /* delete the Environment Variable */
    unsetenv("_EDC_ANSI_OPEN_DEFAULT");

    /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT*/
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("program2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
        (x != NULL) ? x : "undefined");
}
```

### Output

```
program2 _EDC_ANSI_OPEN_DEFAULT = Y
program2 _EDC_ANSI_OPEN_DEFAULT = undefined
```

**Y** The value defined for the environment variable.

### undefined

The environment variable is not defined.

## Related Information

- “getenv—Get Value of Environment Variables” on page 255
- “setenv—Add, Change, or Delete an Environment Variable” on page 463
- “system—Execute a Command” on page 527.

## updateCacheEntry—Add a New or Update an Existing Cache Entry

This function adds a new cache entry or updates an existing cache entry.

### Format

```
#include <c$cach.h>
long      updateCacheEntry(const cacheToken *cache_to_update,
                           const void *primary_key,
                           const long *primary_key_length,
                           const void *secondary_key,
                           const long *secondary_key_length,
                           const long *size_of_entry,
                           const void *entry_data,
                           const long *timeout,
                           const char *invalidateOthers);
```

#### cache\_to\_update

The returned cache\_token from the newCache function that created the logical record cache.

#### primary\_key

A pointer to a field that contains the value for the primary key.

#### primary\_key\_length

The length of the primary key for a cache entry. The length specified must be equal to or less than the maximum value specified for the newCache function.

#### secondary\_key

A pointer to a field that contains the value of the secondary key. If a secondary key was not specified for the newCache function, this field is ignored and will contain a NULL pointer.

#### secondary\_key\_length

The length of the specified secondary key. The length specified must be equal to or less than the maximum value specified for the newCache function.

#### size\_of\_entry

A pointer to a field that contains the length of the passed entry data.

#### entry\_data

A pointer to the area that contains the actual entry data to be saved in the logical record cache.

#### timeout

An integer indicating the number of seconds that a cache entry can reside in cache before it is flushed. Once this value is reached, the cache entry is marked as *not valid* and results in a *not found* condition. This forces a cache update for the cache entry. If a nonzero value is specified, this value overrides the value specified for **castout\_time** for the newCache function.

#### invalidateOthers

A pointer to a char that has been set to either Cache\_NoInvalidate or Cache\_Invalidate. If one of the following conditions is met, the updateCacheEntry function only modifies this processor's cache:

- The pointer is NULL.
- The pointer to a char is set to Cache\_NoInvalidate.
- The cache is processor unique.
- A processor shared cache is processing in local mode.

If the pointer to a char is set to Cache\_Invalidate, logical record cache support tries to invalidate any other processor's cache entry for the specified entry.

## updateCacheEntry

### Normal Return

#### CACHE\_SUCCESS

The function is completed successfully.

### Error Return

One of the following:

#### CACHE\_ERROR\_HANDLE

The cache\_token provided for the cache\_to\_update parameter is not valid.

#### CACHE\_ERROR\_PARAM

The value passed for one of the parameters is not valid.

## Programming Considerations

To ensure data integrity, the caller must ensure that a locking mechanism is used when using the updateCacheEntry function.

## Examples

The following example adds a cache entry to the file system directory cache.

```
#include <c$cach.h>
#include <i$glue.h>

struct ilink * link;
struct TPF_directory_entry new_tde =
    { TPF_FS_DIRECTORY_CURRENT_VERSION };
const long dataLength = sizeof(new_tde); /* size of directory entry */
const long primaryKeyLgh = strlen( link->ilink_file_name );
const long secondaryKeyLgh = sizeof(ino_t); /* INODE number */
struct icontrl * contrl_ptr; /* pointer file system control area */

/* setup file system directory entry with values from the input */

new_tde.TPF_directory_entry_ino =
    link->ilink_file_inode_ordinal;
new_tde.TPF_directory_entry_igen =
    link->ilink_file_inode->inode_igen;

/* get pointer to file system directory cache token if one */

contrl_ptr = cinfc_fast_ss(CINFC_CMMZERO,
    ecbptr()->celdbi );

if (contrl_ptr->icontrl_dcacheToken.token1 != 0)

/* have a directory cache, update it with directory entry */

    updateCacheEntry(&contrl_ptr->icontrl_dcacheToken,
        link->ilink_file_name,
        &primaryKeyLgh,
        &link->ilink_parent_inode->inode_ino,
        &secondaryKeyLgh,
        &dataLength,
        &new_tde,
        NULL,
        NULL );
```



## **Related Information**

- “deleteCache—Delete a Logical Record Cache” on page 81
- “deleteCacheEntry—Delete a Cache Entry” on page 82
- “flushCache—Flush the Cache Contents” on page 198
- “newCache—Create a New Logical Record Cache” on page 376
- “readCacheEntry—Read a Cache Entry” on page 413.

## utime—Set File Access and Modification Times

This function sets file access and modification times.

### Format

```
#define _POSIX_SOURCE
#include <utime.h>
int utime(const char *pathname, const struct utimbuf *newtimes);
```

#### **pathname**

The file whose time stamp will be changed.

#### **newtimes**

A pointer to a structure, `utimbuf`, that specifies the new access and modification times.

This function sets the access and modification times of **pathname** to the values in the `utimbuf` structure. If **newtimes** is a NULL pointer, the access and modification times are set to the current time.

Normally, the user ID (UID) of the calling process must match the owner UID of the file, or the calling process must have appropriate privileges. However, if **newtimes** is a NULL pointer, the UID of the calling process must match the owner UID of the file, or the calling process must have write permission to the file or appropriate privileges.

The contents of a `utimbuf` structure are:

- `time_t` `actime` - The new access time (The `time_t` type gives the number of seconds since the epoch).
- `time_t` `modtime` - The new modification time.

### Normal Return

If successful, the `utime` function returns the value 0 and updates the access and modification times of the file to those specified.

### Error Return

If unsuccessful, the `utime` functions returns the value `-1`, does not update the file times and sets `errno` to one of the following:

- |                     |  |
|---------------------|--|
| <b>EACCES</b>       | The process did not have search permission for some component of the <b>pathname</b> prefix; or all of the following are true: <ul style="list-style-type: none"> <li>• <b>newtimes</b> is NULL.</li> <li>• The user ID of the process does not match the file's owner.</li> <li>• The process does not have write permission on the file.</li> <li>• The process does not have appropriate privileges.</li> </ul> |
| <b>EBUSY</b>        | A loop exists in symbolic links. This error is issued if more than <code>POSIX_SYMLINK</code> symbolic links (defined in the <code>limits.h</code> header file) are detected in the resolution of <b>pathname</b> .  |
| <b>ENAMETOOLONG</b> | The <b>pathname</b> is longer than <code>PATH_MAX</code> characters, or some component of <b>pathname</b> is longer than <code>NAME_MAX</code> characters. For symbolic links, the length of the path name string substituted for a symbolic link exceeds <code>PATH_MAX</code> . The <code>PATH_MAX</code> and <code>NAME_MAX</code> values can be determined using the <code>pathconf</code> function.           |

<b>ENOENT</b>	There is no file names <b>pathname</b> , or the <b>pathname</b> argument is an empty string.
<b>ENOTDIR</b>	Some component of the <b>pathname</b> is not a directory.
<b>EPERM</b>	The <b>newtimes</b> specified is not NULL, the effective user ID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.
<b>EROFS</b>	The <b>pathname</b> specified is on a read-only file system.

## Programming Considerations

None.

## Examples

The following example creates a file, `utime.file`, and then uses the `utime` function to modify the timestamp of the file.

```
#define _POSIX_SOURCE
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <utime.h>

int main(void) {
    int fd;
    char fn[]="utime.file";
    struct utimbuf ubuf;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        ubuf.modtime = 0;
        time(&ubuf.actime);
        if (utime(fn, ubuf) != 0)
            perror("utime() error");
        unlink(fn);
    }
    return 0;
}
```

## Related Information

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## vfprintf–Format and Print Data to a Stream

This function formats and writes data to a stream.

### Format

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, v_list arg_ptr)
```

#### **stream**

The text output file.

#### **format**

The format template.

#### **arg\_ptr**

Replacement arguments for **format**.

The `vfprintf` function is similar to the `fprintf` function except that **arg\_ptr** points to a list of arguments whose number can vary from call to call in the program. These arguments must be initialized by `va_start` for each call. In contrast, the `fprintf` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program. For a specification of the **format** string, see “`fprintf`, `printf`, `sprintf`–Format and Write Data” on page 201.

The `vfprintf` function has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an end-of-file (EOF) has been reached.

### Normal Return

If there is no error, the `vfprintf` function returns the number of characters written to **stream**.

### Error Return

If an error occurs, the `vfprintf` function returns a negative value.

**Note:** In contrast to some UNIX-based implementations of C language, the TPF C library implementation of the `vprintf` function increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the `va_end` macro after each call to the `vsprintf` function.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

### Examples

The following example prints out a variable number of strings to the `myfile.dat` file using `vfprintf`.

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

void vout(FILE *stream, char *fmt, ...);
char fmt1 [] = "%s %s %s\n";
```

```
int main(void)
{
    FILE *stream;
    stream = fopen("myfile.dat", "w");

    vout(stream, fmt1, "Sat", "Sun", "Mon");
}

void vout(FILE *stream, char *fmt, ...)
{
    va_list arg_ptr;

    va_start(arg_ptr, fmt);
    vfprintf(stream, fmt, arg_ptr);
    va_end(arg_ptr);
}
```

**Output**

Sat Sun Mon

**Related Information**

- “vprintf–Format and Print Data to stdout” on page 680
- “vsprintf–Format and Print Data to a Buffer” on page 682.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## vprintf–Format and Print Data to stdout

This function formats and writes data to the standard output stdout stream.

### Format

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg_ptr);
```

#### format

The format template.

#### arg\_ptr

The replacement arguments for format.

The vprintf function is similar to printf except that **arg\_ptr** points to a list of arguments whose number can vary from call to call in the program. These arguments must be initialized by va\_start for each call. In contrast, printf can have a list of arguments, but the number of arguments in that list is fixed when you compile the program. For a specification of the *format* string, see “fprintf, printf, sprintf–Format and Write Data” on page 201.

The vprintf function has the same restriction as any write operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must be an intervening reposition unless an end-of-file (EOF) has been reached.

### Normal Return

If there is no error, vprintf returns the number of characters written to stdout.

### Error Return

If an error occurs, vprintf returns a negative value.

**Note:** In contrast to some UNIX-based implementations of C language, the TPF C library implementation of the vprintf family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the va\_end macro after each call to vprintf.

## Programming Considerations

- The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

## Examples

The following example prints out a variable number of strings to stdout using vprintf.

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

void vout(char *fmt, ...);
char fmt1 [] = "%s %s %s %s %s \n";

int main(void)
{
    vout(fmt1, "Mon", "Tues", "Wed", "Thurs", "Fri");
}
```

```
}  
  
void vout(char *fmt, ...)  
{  
    va_list arg_ptr;  
  
    va_start(arg_ptr, fmt);  
    vprintf(fmt, arg_ptr);  
    va_end(arg_ptr);  
}
```

**Output**

Mon Tues Wed Thurs Fri

**Related Information**

- “vfprintf–Format and Print Data to a Stream” on page 678
- “vsprintf–Format and Print Data to a Buffer” on page 682.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

## vsprintf–Format and Print Data to a Buffer

This function formats and writes data to a buffer.

### Format

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *target-string, const char *format., va_listst arg_ptr);
```

#### target-string

The buffer to which formatted output is to be written.

#### format

The format template.

#### arg\_ptr

The replacement arguments for format.

The `vsprintf` function is similar to the `sprintf` function except that **arg\_ptr** points to a list of arguments whose number can vary from call to call in the program. In contrast, the `sprintf` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program. For a specification of the **format** string, see “`fprintf`, `printf`, `sprintf`–Format and Write Data” on page 201.

### Normal Return

If there is no error, the `vsprintf` function returns the number of characters written to *target-string*.

### Error Return

If an error occurs, the `vsprintf` function returns a negative value.

**Note:** In contrast to some UNIX-based implementations of C language, the TPF C library implementation of the `vprintf` family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the `va_end` macro after each call to the `vsprintf` function.

## Programming Considerations

None.

## Examples

The following example assigns a variable number of strings to `string` and prints the string that is the result using the `vsprintf` function.

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

void vout(char *string, char *fmt, ...);
char fmt1 [] = "%s %s %s\n";

int main(void)
{
    char string[100];

    vout(string, fmt1, "Sat", "Sun", "Mon");
    printf("The string is: %s\n", string);
}

void vout(char *string, char *fmt, ...)
```



```
va_list arg_ptr;  
  
va_start(arg_ptr, fmt);  
vsprintf(string, fmt, arg_ptr);  
va_end(arg_ptr);  
}
```

**Output**

The string is: Sat Sun Mon

**Related Information**

- “vfprintf–Format and Print Data to a Stream” on page 678
- “vprintf–Format and Print Data to stdout” on page 680.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.

---

## wait—Wait for Status Information from a Child Process

This function causes the calling process to be suspended until one of its child processes exits.

### Format

```
#include <sys/wait.h>
int wait(int *stat_loc);
```

#### **stat\_loc**

A pointer to an integer where the `wait` function will return the status of the child process. If the `wait` function returns because the status of a child process is available, the return value will equal the process identifier (ID) of the exiting process. For this, if the value of **stat\_loc** is not NULL, information is stored in the location pointed to by **stat\_loc**. If status is returned from a terminated child process that returned a value of 0, the value stored at the location pointed to by **stat\_loc** will be 0. If the return is greater than 0, you can evaluate this information using the following macros:

```
WEXITSTATUS
WIFEXITED
WIFSIGNALED
WTERMSIG.
```

### Normal Return

If successful, the return code is set to the process ID of the process for which status is available.

### Error Return

If unsuccessful, the `wait` function returns a value of `-1` and sets `errno` to the following:

#### **ECHILD**

The calling process does not have any child processes that need to be handled by the `wait` function.

### Programming Considerations

- If there are any queued signals for the calling ECB to handle, these signals are handled after the status of child processes is checked. Signals received while waiting for child process status are handled as they are received. This means that if the `signal` function is used to install signal handlers, the placement of the `wait` function in the code is subject to the same restrictions as the `raise` function.
- If the `signal` function was coded to ignore SIGCHLD signals, the `wait` function will not return until all of the child processes of the callers have exited, at which time it returns a value of `-1` with `errno` set to ECHILD. Otherwise, the `wait` function returns as soon as a child process exits. If a child process has exited before the `wait` function is called, the `wait` function returns immediately.

### Examples

The following example creates a child process and then calls the `wait` function to ensure that the child process was completed successfully.

```
#include <sysapi.h>
#include <signal.h>
#include <sys/wait.h>
```

```

:
:
pid_t child_pid;
pid_t rc_pid;
int chld_state;
/* Create a child process*/
:
child_pid = tpf_fork(&create_parameters);
:
/* Check status of child process*/
rc_pid = wait( &chld_state );
if (rc_pid > 0)
{
    if (WIFEXITED(chld_state)) {
        printf("Child exited with RC=%d\n",WEXITSTATUS(chld_state));
    }
    if (WISIGNALLED(chld_state)) {
        printf("Child exited via signal %d\n",WTERMSIG(chld_state));
    }
}
else
/* if no PID returned, then an error */
{
    if (errno == ECHILD) {
        printf("No children exist.\n");
    }
    else {
        printf("Unexpected error.\n");
        abort();
    }
}
:
:

```

## Related Information

- “alarm—Schedule an Alarm” on page 13
- “kill—Send a Signal to a Process” on page 298
- “signal—Install Signal Handler” on page 495
- “tpf\_fork—Create a Child Process” on page 594
- “tpf\_process\_signals—Process Outstanding Signals” on page 622
- “waitpid—Obtain Status Information from a Child Process” on page 687
- “WEXITSTATUS—Obtain Exit Status of a Child Process” on page 690
- “WIFEXITED—Query Status to See If a Child Process Ended Normally” on page 694
- “WIFSIGNALED—Query Status to See If a Child Process Ended Abnormally” on page 695
- “WTERMSIG—Determine Which Signal Caused the Child Process to Exit” on page 699.

---

## waitc—Wait For Outstanding I/O Completion

This function suspends processing of an entry until all I/O requests associated with the ECB are complete. Error information is returned, if applicable.

### Format

```
#include <tpfio.h>
int      waitc(void);
```

### Normal Return

Integer value of zero.

### Error Return

Integer value representing the value of CE1SUG (gross level error indicator byte).

## Programming Considerations

- For errors related to I/O hardware, a storage dump is issued and CRAS is informed of the condition.
- Certain output operations (such as file type functions) may not be completed following the execution of the waitc function.

## Examples

The following example calls the waitc function to ensure that all outstanding I/O have completed, then issues a message and aborts if the waitc function indicates an error.

```
#include <tpfio.h>
...
if(waitc())
{
    serrc_op(SERRC_EXIT,0x1234,"ERROR SYNCHRONIZING VPH TAPE",NULL) ;
}
```

## Related Information

- “filnc—File a Record with No Release” on page 165
- “findc—Find a Record” on page 174
- “finhc—Find and Hold a File Record” on page 185
- “toutc—Write Real-Time Tape Record” on page 558
- “tprdc—Read General Tape Record” on page 641
- “trewc—Rewind General Tape and Wait” on page 643
- “tsync—Synchronize a Tape” on page 645.

## waitpid—Obtain Status Information from a Child Process

This function allows the calling process to obtain the exit status information from a child process.

Options permit the caller to either suspend execution of the calling process until a child process ends or return immediately.

### Format

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *stat_loc,
              int options);
```

#### pid

A set of child processes for which status is requested. Only one status is returned per waitpid function call.

- If **pid** is equal to `-1`, status is requested for any child process. If status information is available for two or more processes, the order in which their status is reported is not specified.
- If **pid** is greater than 0, status is requested for a single process.
- If **pid** is equal to 0, status is requested for any process whose process group ID (GID) matches the process group ID of the caller.
- If **pid** is less than `-1`, status is requested for any process whose process group ID is equal to the absolute value of **pid**.

#### stat\_loc

A pointer to an integer where the wait function will return the status of the child process. If the waitpid function returns because a process has exited, the return value is equal to the **pid** of the exiting process. For this, if the value of **stat\_loc** is not NULL, information is stored in the location pointed to by **stat\_loc**. If status is returned from a terminated child process that returned a value of 0, the value stored at the location pointed to by **stat\_loc** is 0. If the return is greater than 0, you can evaluate this information using the following macros:

```
WEXITSTATUS
WIFEXITED
WIFSIGNALED
WTERMSIG.
```

#### options

0 or the following flag:

##### WNOHANG

The waitpid function does not suspend the calling process if status is not immediately available for any process specified by the **pid** parameter.

### Normal Return

The normal return is one of the following:

- If the status of a process is available, the return code is set to the **pid** of the process for which status is available.
- If **WNOHANG** is specified and there are valid processes to report on, but there is no status available, a value of 0 is returned.

## waitpid

- If **WNOHANG** is not specified and there are valid processes to report on, but there is no status available, the calling process is blocked until status becomes available, at which time the return code is set to the **pid** of the process for which status is available.

## Error Return

If unsuccessful, the waitpid function returns a value of **-1** and sets **errno** to one of the following:

- |               |   |
|---------------|---|
| <b>ECHILD</b> | The process specified by the <b>pid</b> parameter does not exist. |
| <b>EINVAL</b> | The value of the <b>options</b> argument is not valid.            |

## Programming Considerations

- If there are any queued signals for the calling ECB to handle, these signals are handled after the status of child processes is checked. Signals that are received while waiting for child process status are handled as they are received. This means that if the **signal** function was used to install signal handlers, the placement of the waitpid function in the code is subject to the same restrictions as the **raise** function.
- If the waitpid function is coded to wait for a specific child process to exit and while the parent is waiting for that process, a different child process exits first; the exit status of the first child process is maintained for the parent process to query with a subsequent **wait** or waitpid call.
- If the **signal** function was coded to ignore **SIGCHLD** signals, exit status of the exiting child processes is not available. The waitpid function will return immediately without status if **WNOHANG** is specified. If **WNOHANG** is not specified, the waitpid function will not return until all child processes have exited, at which time it returns with a value of **-1** and **errno** is set to **ECHILD**.

## Examples

The following example shows how the waitpid function is used to wait for a single child process to exit and query its exit status.

```
#include <sysapi.h>
#include <signal.h>
#include <sys/wait.h>
:
:

pid_t child_pid;
pid_t rc_pid;
int chld_state;
int my_opt=WNOHANG;
:
:
/* Create a child process*/
child_pid = tpf_fork(&create_parameters);
:
:
/* Check status of child process*/
rc_pid = waitpid( child_pid, &chld_state, WNOHANG);
if (rc_pid > 0)
{
    if (WIFEXITED(chld_state)) {
        printf("Child exited with RC=%d\n",WEXITSTATUS(chld_state));
    }
    if (WISIGNALLED(chld_state)) {
        printf("Child exited via signal %d\n",WTERMSIG(chld_state));
    }
}
else
```

```

/* If no error, continue*/
{
    if (rc_pid < 0) {
        if (errno == ECHILD) {
            printf("Child does not exist\n");
        }
        else {
            printf("Bad argument passed to waitpid\n");
            abort();
        }
    }
}
:

```

## Related Information

- “alarm—Schedule an Alarm” on page 13
- “kill—Send a Signal to a Process” on page 298
- “signal—Install Signal Handler” on page 495
- “sleep—Suspend the Calling Process” on page 507
- “tpf\_fork—Create a Child Process” on page 594
- “tpf\_process\_signals—Process Outstanding Signals” on page 622
- “wait—Wait for Status Information from a Child Process” on page 684
- “WEXITSTATUS—Obtain Exit Status of a Child Process” on page 690
- “WIFEXITED—Query Status to See If a Child Process Ended Normally” on page 694
- “WIFSIGNALED—Query Status to See If a Child Process Ended Abnormally” on page 695
- “WTERMSIG—Determine Which Signal Caused the Child Process to Exit” on page 699.

### WEXITSTATUS—Obtain Exit Status of a Child Process

This macro queries the child termination status provided by the `wait` and `waitpid` functions. If the `WIFEXITED` macro indicates that the child process exited normally, the `WEXITSTATUS` macro returns the exit code specified by the child process.

#### Format

```
#include <sys/wait.h>
int WEXITSTATUS(int status);
```

##### **status**

The status field that was filled in by the `wait` or `waitpid` function.

#### Normal Return

The `WEXITSTATUS` macro is always successful.

If the `WIFEXITED` macro indicates that the child process exited normally, the `WEXITSTATUS` macro returns the exit code specified by the child. If the `WIFEXITED` macro indicates that the child process did not exit normally, the value returned by the `WEXITSTATUS` macro has no meaning.

#### Error Return

Not applicable.

#### Programming Considerations

None.

#### Examples

See “`wait`—Wait for Status Information from a Child Process” on page 684 for an example of the `WEXITSTATUS` macro.

#### Related Information

- “`wait`—Wait for Status Information from a Child Process” on page 684
- “`waitpid`—Obtain Status Information from a Child Process” on page 687
- “`WIFEXITED`—Query Status to See If a Child Process Ended Normally” on page 694
- “`WIFSIGNALED`—Query Status to See If a Child Process Ended Abnormally” on page 695
- “`WTERMSIG`—Determine Which Signal Caused the Child Process to Exit” on page 699.



## wgtac—Locate Terminal Entry

This function returns information from the WGTA table concerning a terminal LNIATA (or pseudo-LNIATA) value.

### Format

```
#include <tpfapi.h>
struct wg0ta *wgtac(int lniata, unsigned char cpuid, struct wg0ta *storage);
```

#### lniata

An integer representing the LNIATA (or SNA RID) for which information is desired. Only 24 bits of this integer are considered significant.

#### cpuid

An unsigned character representing the hexadecimal CPU identification of the CPU on which the calling ECB is running.

#### storage

A pointer to a free working storage area large enough to hold a completed wg0ta structure. This address is overwritten with a completed wg0ta, and is passed back as the return value if the search for the requested LNIATA is successful. The wg0ta structure is defined in tpfapi.h.

### Normal Return

Pointer to struct wg0ta for the requested LNIATA.

### Error Return

NULL pointer.

### Programming Considerations

None.

### Examples

The following example obtains a copy of the WGTA table information for the LNIATA referred to from the issuing ECB.

```
#include <tpfapi.h>
struct wg0ta *wgta;
:
wgta = (struct wg0ta *) getcc(D4, TYPE, L1);
if(!(wgta = wgtac(ecbptr()->ebrout,ecbptr()->celcpd,wgta)))
{
    /* Item not found if return was zero */
    puts("LNIATA not referenced in WGTA table\n");
    relcc(D4);
}
else
    /*      Normal processing path      */
:
:
```

### Related Information

- Data macro WG0TA
- “wgtac\_ext—Locate Terminal Entry with Extended Options” on page 692.

## wgtac\_ext—Locate Terminal Entry with Extended Options

This function returns information from the terminal address table (WGTA) about a line number, interchange address, and terminal address (LNIATA) or pseudo-LNIATA value.

### Format

```
#include <tpfapi.h>
struct wg0ta *wgtac_ext(int lniata, unsigned char cpuid,
                       struct wg0ta *storage, unsigned int ext);
```

#### **lniata**

An integer representing the LNIATA or Systems Network Architecture (SNA) resource identifier (RID) for which information is desired. Only 24 bits of this integer are considered significant.

#### **cpuid**

An unsigned character representing the hexadecimal central processing unit (CPU) identification of the CPU on which the calling entry control block (ECB) is running.

#### **storage**

A pointer to a free working storage area large enough to hold a completed wg0ta structure. This address is overwritten with a completed wg0ta and is passed back as the return value if the search for the requested LNIATA is successful and WGTA\_NO\_UPDATE is specified. If WGTA\_UPDATE is specified, this parameter should be set to zero. The wg0ta structure is defined in the tpfapi.h header file.

#### **ext**

Use one of the following options defined in the tpfapi.h header file to indicate if the entry will be updated:

##### **WGTA\_NO\_UPDATE**

Specifies that no updates will be performed. The WGTA entry will be copied into the storage location pointed to by the storage parameter.

##### **WGTA\_UPDATE**

Specifies that updates may be performed. A pointer to the actual WGTA table entry is returned.

### Normal Return

A pointer to struct wg0ta for the requested LNIATA.

### Error Return

NULL pointer.

### Programming Considerations

When specifying WGTA\_UPDATE, ensure that you have the protect key set to 0 before attempting to update any fields in the WGTA entry; otherwise, a system error will be generated.

### Examples

The following example obtains a pointer to the WGTA entry for the LNIATA referred to from the issuing ECB.

```

#include <tpfapi.h>
struct wg0ta *wgta;
:
if(!(wgta = wgtac_ext(ecbptr()->ebrout,ecbptr()->celcpd,0,WGTA_UPDATE)))
{
    /* Item not found if return was zero */
    puts("LNIATA not referenced in WGTA table\n");
}
else
    /*      Normal processing path      */
:

```

## Related Information

“wgtac—Locate Terminal Entry” on page 691.

---

## **WIFEXITED—Query Status to See If a Child Process Ended Normally**

This macro queries the child termination status provided by the `wait` and `waitpid` functions, and determines whether the child process ended normally.

### **Format**

```
#include <sys/wait.h>
int WIFEXITED(int status);
```

#### **status**

The status field that was filled in by the `wait` or `waitpid` function.

### **Normal Return**

The `WIFEXITED` macro is always successful.

If the child process for which status was returned by the `wait` or `waitpid` function exited normally, the `WIFEXITED` macro evaluates to `TRUE` and the `WEXITSTATUS` macro is used to query the exit code of the child process. Otherwise, the `WIFEXITED` macro evaluates to `FALSE`.

### **Error Return**

Not applicable.

### **Programming Considerations**

None.

### **Examples**

See “`wait`—Wait for Status Information from a Child Process” on page 684 for an example of the `WIFEXITED` macro.

### **Related Information**

- “`wait`—Wait for Status Information from a Child Process” on page 684
- “`waitpid`—Obtain Status Information from a Child Process” on page 687
- “`WEXITSTATUS`—Obtain Exit Status of a Child Process” on page 690
- “`WIFSIGNALED`—Query Status to See If a Child Process Ended Abnormally” on page 695
- “`WTERMSIG`—Determine Which Signal Caused the Child Process to Exit” on page 699.

---

## WIFSIGNALED—Query Status to See If a Child Process Ended Abnormally

This macro queries the child termination status provided by the `wait` and `waitpid` functions. It determines if the child process exited because it raised a signal that caused it to exit. It is also used with the `WTERMSIG` macro to determine if the child process exited because of a system error.

### Format

```
#include <sys/wait.h>
int WIFSIGNALED(int status);
```

#### **status**

The status field that was filled in by the `wait` or `waitpid` function.

### Normal Return

The `WIFSIGNALED` macro is always successful.

If the child process for which status was returned by the `wait` or `waitpid` function exited because it raised a signal that caused it to exit, the `WIFSIGNALED` macro evaluates to `TRUE` and the `WTERMSIG` macro can be used to determine which signal was raised by the child process. Otherwise, the `WIFSIGNALED` macro evaluates to `FALSE`.

**Note:** See “`WTERMSIG`—Determine Which Signal Caused the Child Process to Exit” on page 699 for more information about how to determine if the child process exited because of a system error.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

See “`wait`—Wait for Status Information from a Child Process” on page 684 for an example of the `WIFSIGNALED` macro.

### Related Information

- “`wait`—Wait for Status Information from a Child Process” on page 684
- “`waitpid`—Obtain Status Information from a Child Process” on page 687
- “`WEXITSTATUS`—Obtain Exit Status of a Child Process” on page 690
- “`WIFEXITED`—Query Status to See If a Child Process Ended Normally” on page 694
- “`WTERMSIG`—Determine Which Signal Caused the Child Process to Exit” on page 699.

## write—Write Data to a File Descriptor

This function writes data to a file.

### Note

This description applies only to files. See the *TPF Transmission Control Protocol/Internet Protocol* for more information about the write function for sockets.

### TPF deviation from POSIX

The TPF system does not support the `ctime` (change time) time stamp.

## Format

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t N);
```

### fildes

The file descriptor.

### buf

The data that will be written.

**N** The number of bytes of data to be written.

This function writes **N** bytes from **buf** to the file associated with **fildes**. **N** must not be greater than `LONG_MAX` (defined in the `limits.h` header file). If **N** is zero, the write function returns a value of zero without attempting any other action.

If **fildes** refers to a regular file or any other type of file on which a process can seek, the write function begins writing at the file offset associated with **fildes**. A successful write function increments the file offset by the number of bytes written. If the incremented file offset is greater than the previous length of the file, the length of the file is set to the new file offset.

If **fildes** refers to a file on which a process cannot seek, the write function begins writing at the current position. There is no file offset associated with such a file.

If `O_APPEND` (defined in the `fcntl.h` header file) is set for the file, the write function sets the file offset to the end of the file before writing the output.

If there is not enough room to write the requested number of bytes (for example, because there is not enough room on the disk), the write function writes as many bytes as the remaining space can hold.

For character special files that support nonblocking writes and cannot accept data immediately, the effect is one of the following:

- If `O_NONBLOCK` is not set, the write function blocks the process until the data can be written.
- If `O_NONBLOCK` is set, the write function does not block the process. If some data can be written without blocking the process, the write function writes what it can and returns the number of bytes written. Otherwise, it sets the `errno` function to `EAGAIN` and returns a value of `-1`.

A successful write function updates the modification time for the file.

## Normal Return

If successful, the write function returns the number of bytes written, less than or equal to **N**.

## Error Return

If unsuccessful, the write function returns a **-1** value and sets the errno function to one of the following:

<b>EAGAIN</b>	O_NONBLOCK is set and output cannot be written immediately.
<b>EBADF</b>	<b>fildes</b> is not a valid open file descriptor.
<b>EFBIG</b>	Writing to the output file would exceed the maximum file size supported by the implementation.
<b>EINTR</b>	A signal interrupted write function processing before any output was written.
<b>ENOSPC</b>	There is no available space left on the output device.
<b>EPIPE</b>	There was an attempt to write to a pipe that is not open for reading. A SIGPIPE signal is generated.

## Programming Considerations

The TPF system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

## Examples

The following example writes 1 000 000 bytes to a file.

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

#define mega_string_len 1000000

main() {
    char *mega_string;
    int fd, ret;
    char fn[]="write.file";

    if ((mega_string = (char*) malloc(mega_string_len)) == NULL)
        perror("malloc() error");
    else if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        memset(mega_string, '0', mega_string_len);
        if ((ret = write(fd, mega_string, mega_string_len)) == -1)
            perror("write() error");
        else printf("write() wrote %d bytes\n", ret);
        close(fd);
        unlink(fn);
    }
}
```

### Output

```
write() wrote 1000000 bytes
```

**write**

## **Related Information**

- “creat—Create a New File or Rewrite an Existing File” on page 54
- “dup—Duplicate an Open File Descriptor” on page 97
- “fcntl—Control Open File Descriptors” on page 129
- “fwrite—Write Items” on page 239
- “lseek—Change the Offset of a File” on page 315
- “open—Open a File” on page 380
- “read—Read from a File” on page 410.

See Appendix E, “Programming Support for the TPF File System” on page 1405 for more information about TPF File System C Functions.



---

## WTERMSIG—Determine Which Signal Caused the Child Process to Exit

This macro queries the termination status of a child process to determine which signal caused the child process to exit. Status is provided by the `wait` and `waitpid` functions.

### Format

```
#include <sys/wait.h>
int WTERMSIG(int status);
```

#### **status**

The status field that was filled in by the `wait` or `waitpid` function.

### Normal Return

The `WTERMSIG` macro is always successful.

If the `WIFSIGNALED` macro indicates that the child process exited because it raised a signal, the `WTERMSIG` macro returns the numeric value of the signal that was raised by the child process. Signal values are defined in the `sys/signals.h` C header file. A signal value of `SIGILL` means the child process exited because of a system error rather than the raising of a particular signal. If the `WIFEXITED` macro indicates that the child process did not exit because it raised a signal, the value returned by the `WTERMSIG` macro has no meaning.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

See “`wait`—Wait for Status Information from a Child Process” on page 684 for an example of the `WTERMSIG` macro.

### Related Information

- “`wait`—Wait for Status Information from a Child Process” on page 684
- “`waitpid`—Obtain Status Information from a Child Process” on page 687
- “`WEXITSTATUS`—Obtain Exit Status of a Child Process” on page 690
- “`WIFEXITED`—Query Status to See If a Child Process Ended Normally” on page 694
- “`WIFSIGNALED`—Query Status to See If a Child Process Ended Abnormally” on page 695.

## wtopc—Send System Message

This function sends a message to the system operator or any other CRAS terminal. Chaining of messages and an optional header are additional features offered by this function.

### Format

```
#include <tpfapi.h>
int wtopc(const char * text_ptr, long int rout,
          enum t_wtopc_chain chain, const void *header_ptr);
```

#### text\_ptr

Pointer to the message to be sent.

#### rout

Integer containing the addition of routing destinations as defined in `tpfapi.h`. Valid routing destinations are `WTOPC_EBROUT`, `WTOPC_RO`, `WTOPC_PRC`, `WTOPC_TAPE`, `WTOPC_DASD`, `WTOPC_COMM`, `WTOPC_AUDT`.

`WTOPC_UNSQL` can be coded to send unsolicited messages to remote terminals specified in `EBROUT`. (Both `WTOPC_EBROUT` and `WTOPC_UNSQL` have to be coded.) An additional routing destination receives the message as a solicited message.

0 can be entered to indicate that the `WTOPC_EBROUT` value is to be used.

#### chain

This argument must belong to the enumeration type `t_wtopc_chain`, defined in `tpfapi.h`, specifying ONLY 1 of the following values:

##### **WTOPC\_NO\_CHAIN**

Specifies that the current WTOPC message should *not* be chained to a previous WTOPC message.

##### **WTOPC\_CHAIN**

Specifies that current WTOPC message should be chained to a previous WTOPC message.

##### **WTOPC\_CHAIN\_END**

Specifies that current WTOPC message is the end of a chain of messages.

##### **WTOPC\_NO\_PAGE**

Similar to `WTOPC_NO_CHAIN` in that no chaining or paging will be used.

##### **WTOPC\_PAGE**

Specifies that the current WTOPC message will be chained and will use the page support for large messages. Existing paged messages that are being controlled by this ECB are terminated.

##### **WTOPC\_PAGE\_END**

Similar to `WTOPC_CHAIN_END` in that it signals the service routine to end the current message chain.

#### header\_ptr

If a header is desired, this argument points to a header structure as defined in `tpfapi.h`. If a header is not desired, `WTOPC_NO_HEADER`, defined in `tpfapi.h`, can be supplied as the argument. The fields in the header structure are defined as follows:

##### **wtopc\_prefix\_pointer**

char pointer to a 4-character prefix in the header. This field has a default (NULL entered) of the name of the program which called the `wtopc` function.

**wtopc\_number**

short int number field in the header from 0–9999. This field has no default value except when the whole header is defaulted, as explained below.

**wtopc\_letter**

char letter field in the header. This field has a default (\0 entered) of the letter l.

**wtopc\_time**

This field must belong to the enumeration type `t_wtopc_time`, defined in `tpfapi.h`, specifying one of the following times:

**WTOPC\_SYS\_TIME**

Specifies system time should be included in the header.

**WTOPC\_SUBSYS\_TIME**

Specifies subsystem time should be included in the header.

**WTOPC\_NO\_TIME**

Specifies that no time should be included in the header.

This field has a default (0 entered) of `WTOPC_SUBSYS_TIME`.

This parameter has a default (NULL entered) of all the individual default fields, as defined previously, with the number 1 used as the default for the number field.

## Normal Return

Return Codes:

- 0 WTOPC\_SUCCESS - Normal return code. No special action should be taken.
- 1 WTOPC\_PAGE\_STOP - The caller should send no more lines or pages. Either the timeout value for ZPAGE was reached or another long message was sent.
- 2 WTOPC\_NEXT\_PAGE - The caller is authorized to send more output that will follow the previous page.

## Error Return

Not applicable.

## Programming Considerations

- `text_ptr` must point to a message that is NULL terminated whose length is no greater than 255 bytes.
- There are defaults for all the different parts of the header, as defined above, except for the number field when the whole header is NOT defaulted. In this case the number field MUST be supplied.
- `wtopc_insert_header` and `wtopc_routing_list` are functions available, defined in `tpfapi.h`, that simplify the coding of the parameter list for the `wtopc` function.
- `WTOPC_TEXT`, defined in `tpfapi.h`, is an alternate way to call this function when the parameter list consists of an address of the text and the rest NULLs for the defaults.
- If `WTOPC_PAGE` is specified, then the `ROUT` parameter must contain at least `EBROUT`.
- `WTOPC_UNSQL` and `WTOPC_CHAIN` are mutually exclusive. If either `WTOPC_CHAIN` or `WTOPC_CHAIN_END` is specified, `WTOPC_UNSQL` cannot be specified.

## wtopc

- The message text can contain the following number of characters:
  - When UNSOL=NO
    - If a message header is specified, a maximum of 235 characters.
    - If no message header is specified, a maximum of 255 characters.
  - When UNSOL=YES
    - If a message header is specified, a maximum of 161 characters.
    - If no message header is specified, a maximum of 180 characters.
  - WTOPC\_PAGE and WTOPC\_UN SOL are mutually exclusive.

### TARGET(TPF) to ISO-C migration consideration

The wtopc function is implemented for TARGET(TPF) as a macro that type casts its arguments, and therefore accepts mistyped arguments that have the same bit patterns or arithmetic values as the required parameter values. For ISO-C, wtopc is implemented as a function, so the types of its arguments must match the documented interface. TARGET(TPF) code that might execute correctly despite passing mistyped arguments to wtopc will not compile for ISO-C.

## Examples

The following example sends a message to EBROUT with a header and without chaining. The example uses the WTOPC\_INSERT\_HEADER macro.

```
#include <tpfapi.h>
...
struct wtopc_header header_buffer;
...
wtopc_insert_header(&header_buffer, "C000", 2, 'A', WTOPC_SYS_TIME);
wtopc("msg1", WTOPC_EBROUT, WTOPC_NO_CHAIN, &header_buffer);
...
```

The following example sends 2 messages to EBROUT and the PRIME CRAS, chaining the 2 messages together and using the default header.

```
#include <tpfapi.h>
...
wtopc("msg2", WTOPC_EBROUT + WTOPC_PRC, WTOPC_CHAIN, NULL);
wtopc("msg3", WTOPC_EBROUT + WTOPC_PRC, WTOPC_CHAIN_END, NULL);
...
```

The following example sends 2 messages to EBROUT, chaining the 2 messages together, using the default header and using the multi-page support.

```
#include <tpfapi.h>
...
rc = wtopc("msg2", WTOPC_EBROUT, WTOPC_PAGE, NULL);
if (rc == WTOPC_PAGE_STOP)
{
    return(rc);
}
else
{
    /* If it is WTOPC_NEXT_PAGE OR WTOPC_SUCCESS we would want to */
    /* send the next line.                                         */
    wtopc("msg3", WTOPC_EBROUT, WTOPC_PAGE_END, NULL);
}
```

```

}
:

```

The following example sends a message to EBROUT and the PRIME CRAS using the wtopc\_routing\_list macro and formatting the message using the sprintf function. No header is requested.

```

#include <tpfapi.h>
:
long int rout_buffer;
char    message_buffer[100];
char *   s = "msg";
:
sprintf(message_buffer, "%s %d\n", s, 4);
wtopc_routing_list(rout_buffer, WTOPC_EBROUT + WTOPC_PRC);
wtopc(message_buffer, rout_buffer, WTOPC_NO_CHAIN, WTOPC_NO_HEADER);
:

```

The following example sends an unsolicited message to the terminal specified in EBROUT and to RO solicited.

```

#include <tpfapi.h>
:
wtopc("msg5", WTOPC_UNSQL + WTOPC_EBROUT + WTOPC_RO, NULL, NULL);
:

```

## Related Information

- “wtopc\_insert\_header–Save Header for wtopc” on page 704
- “wtopc\_routing\_list–Save Routing List for wtopc” on page 706
- “wtopc\_text–Send System Message” on page 707.

## wtopc\_insert\_header—Save Header for wtopc

This function inserts a header, in correct format for the wtopc function to use, into a 12-byte field pointed to by the **buffer\_ptr** parameter.

### Format

```
#include <tpfapi.h>
void wtopc_insert_header(char * buffer_ptr, char * prefix_ptr,
                        short int number, char letter,
                        enum t_wtopc_time time);
```

#### **buffer\_ptr**

Storage location of where to store the header information.

#### **prefix\_ptr**

char pointer to a 4-character prefix in the header. This field has a default (NULL entered) of the name of the program that called the wtopc function.

#### **number**

short int number field in the header from 0—9999.

#### **letter**

char letter field in the header. NULL can be entered to indicate a default value is to be used by wtopc; the default is the letter 'I'. IBM limits the letters to I, E, and W, and provides equates for these values. You can choose to hardcode a different letter for your messages.

*Table 35. wtopc\_insert\_header letter parameter*

Value Name	Letter	Description
# define WTOPC_INFO	I	Informational message
#define WTOPC_WARNING	W	Attention message
#define WTOPC_ERROR	E	Error message

#### **time**

This field must belong to the enumeration type t\_wtopc\_time, defined in tpfapi.h, specifying one of the following times:

##### **WTOPC\_SYS\_TIME**

Specifies system time should be included in the header.

##### **WTOPC\_SUBSYS\_TIME**

Specifies subsystem time should be included in the header.

##### **WTOPC\_NO\_TIME**

Specifies that no time should be included in the header.

This field has a default (NULL entered) of WTOPC\_SUBSYS\_TIME.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

This function is designed to simplify parameter coding for the wtopc function.

## Examples

The following example sets up a header for the wtopc function to use.

```
#include <tpfapi.h>
:
:
struct wtopc_header header_buffer;
:
:
wtopc_insert_header(&header_buffer, "C000", 2, 'A', WTOPC_SYS_TIME);
:
:
```

The following example sets up a header for the wtopc function to use. A NULL is entered as one of the parameters.

```
#include <tpfapi.h>
:
:
struct wtopc_header header_buffer;
:
:
wtopc_insert_header(&header_buffer, NULL, 2, 'A', WTOPC_SYS_TIME);
:
:
```

## Related Information

- “wtopc–Send System Message” on page 700
- “wtopc\_routing\_list–Save Routing List for wtopc” on page 706
- “wtopc\_text–Send System Message” on page 707.

---

## wtopc\_routing\_list—Save Routing List for wtopc

This function inserts the addition of the routing destinations indicated, in correct format for the wtopc function to use, into a field indicated in the parameter list.

### Format

```
#include <tpfapi.h>
void      wtopc_routing_list(long int rout, long int codes);
```

#### **rout**

Storage location for the routing list information.

#### **codes**

Integer containing the addition of routing destinations as defined in tpfapi.h. Valid routing destinations are WTOPC\_EBROUT, WTOPC\_RO, WTOPC\_PRC, WTOPC\_TAPE, WTOPC\_DASD, WTOPC\_COMM, WTOPC\_AUDT, WTOPC\_UNSQL. NULL can be entered to indicate that a WTOPC\_EBROUT value will be used.

### Normal Return

Void.

### Error Return

Not applicable.

## Programming Considerations

This function is designed to simplify parameter coding for the wtopc function.

### Examples

The following example sets up a routing list for the wtopc function to use. EBROUT and the read-only CRAS are indicated in the list.

```
#include <tpfapi.h>
:
:
long int  rout_list;
:
:
wtopc_routing_list(rout_list, WTOPC_EBROUT + WTOPC_RO);
:
:
```

### Related Information

- “wtopc\_insert\_header—Save Header for wtopc” on page 704
- “wtopc—Send System Message” on page 700
- “wtopc\_text—Send System Message” on page 707.



---

## wtopc\_text—Send System Message

This function sends a message to the address contained in EBROUT. This is a simplified way to send a message using the wtopc function when everything except the message address defaults. See “wtopc—Send System Message” on page 700 for more information about the defaults.

### Format

```
#include <tpfapi.h>
void      wtopc_text(const char * text_ptr);
```

**text\_ptr**

Pointer to the message to be sent.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

- text\_ptr must point to a message that is NULL terminated whose length is no greater than 255 bytes.
- No chaining takes place and a default header is used.

### Examples

The following example sends a message to the address contained in EBROUT.

```
#include <tpfapi.h>
:
:
wtopc_text("msg1");
:
```

### Related Information

- “wtopc\_insert\_header—Save Header for wtopc” on page 704
- “wtopc\_routing\_list—Save Routing List for wtopc” on page 706
- “wtopc—Send System Message” on page 700.

## xa\_commit—Commit Work Done for a Transaction Branch

The transaction manager (TM) calls the `xa_commit` function to commit work associated with the transaction branch. Any changes made to resources held during the transaction branch are made permanent.

### Format

```
#include (i$tmcr.h)
int      xa_commit(XID *xid, int rmid, long flags);
```

#### **xid**

A pointer to an exchange identification (XID) structure. XID is a unique identifier assigned by the transaction manager for each transaction branch.

#### **rmid**

An integer, assigned by the TM, that can be used to uniquely identify the called resource manager instance.

#### **flags**

##### **TMNOFLAGS**

Used when no other flags are set.

### Normal Return

<b>XA_OK</b>	Normal completion.
--------------	--------------------

### Error Return

<b>XA_RBROLLBACK</b>	The resource manager (RM) did not commit the work done for the transaction branch. The resource manager has rolled back the work of the branch and has released all held resources.
<b>XAER_RMERR</b>	An error occurred in committing the work performed for the transaction branch and the work of the branch has been rolled back. This error signals a catastrophic event to a transaction manager because other resource managers may successfully commit their work for this branch. This error is returned only when a resource manager concludes that it can never commit the branch and that it cannot hold the resource of the branch in a prepared state.
<b>XAER_RMFAIL</b>	An error occurred that makes the resource manager unavailable. The specified <b>xid</b> may or may not have been prepared.
<b>XAER_NOTA</b>	The specified <b>XID</b> is not known by the resource manager.
<b>XAER_INVAL</b>	Incorrect arguments were specified.
<b>XAER_PROTO</b>	The routine was called incorrectly.

### Programming Considerations

- You must have authorization to issue a restricted C function.
- All transaction branches must have been ended by using the `xa_end()` function with the `TMSUCCESS` flag.

- A transaction manager must issue a separate `xa_commit` function for each transaction branch that accesses the resource manager.
- From the perspective of the resource managers, pointer **xid** is valid only for the duration of the call to the `xa_commit()` function; that is, once the function ends, the transaction manager is permitted to invalidate where **xid** points. Resource managers are encouraged to use private copies of **\*xid** after the function ends.

## Examples

```
#include (i$tmcr.h)

    if (xa_commit(XID *xid, int rmid, long flags) != XA_OK)
    {
        /* handle error condition */
        :
    }
    else {
        /* continue with normal processing */
        :
    }
```

## Related Information

- “`xa_start`—Start Work for a Transaction Branch” on page 720
- “`xa_open`—Open a Resource Manager” on page 712.

## xa\_end—End Work Performed for a Transaction Branch

The transaction manager (TM) calls the `xa_end` function when an entry control block (ECB) is completed, or needs to suspend work on, a transaction branch. When the `xa_end()` function successfully returns, the calling ECB is no longer actively associated with the branch, but the branch still exists.

A resource manager (RM) will mark the transaction branch as rollback-only and return `XA_RBROLLBACK` if the **TMFAIL** flag is set.

### Format

```
#include (<xtmcr.h>)
int      xa_end(XID *xid, int rmid, long flags);
```

#### **xid**

A pointer to an XID structure. XID is a unique identifier assigned by the transaction manager for each transaction branch.

#### **rmid**

An integer, assigned by the TM, that can be used to uniquely identify the called resource manager instance.

#### **flags**

##### **TMSUSPEND**

Used by the `tx_suspend_tpf` function.

##### **TMSUCCESS**

A resource manager has not returned `XA_RBROLLBACK` on any previous XA function calls for this transaction branch.

##### **TMFAIL**

A resource manager has returned `XA_RBROLLBACK` on previous XA function calls for this transaction branch.

### Normal Return

<b>XA_OK</b>	Normal completion.
--------------	--------------------

### Error Return

<b>XA_RBROLLBACK</b>	The resource manager has disassociated the transaction branch from the thread of control and has marked as rollback-only the work performed for the transaction branch.
<b>XAER_RMERR</b>	An error occurred in disassociating the transaction branch from the thread of control.
<b>XAER_RMFAIL</b>	An error occurred that makes the resource manager unavailable.
<b>XAER_NOTA</b>	The specified XID is not known by the resource manager.
<b>XAER_INVAL</b>	Incorrect arguments were specified.
<b>XAER_PROTO</b>	The routine was called incorrectly.

### Programming Considerations

- You must have authorization to issue a restricted C function.

- From the resource manager's perspective, the pointer **xid** is valid only for the duration of the call to the `xa_end()` function. That is, once the function completes, the transaction manager is permitted to invalidate where `xid` points. Resource managers are encouraged to use private copies of **\*xid** after the function ends.

## Examples

```
#include (i$tmcr.h)

    if (xa_end(XID *xid, int rmid, long flags) != XA_OK)
    {
        /* handle error condition */
        :
    }
    else {
        /* continue with normal processing */
        :
    }
```

## Related Information

- “`xa_start`—Start Work for a Transaction Branch” on page 720
- “`xa_open`—Open a Resource Manager” on page 712.

## xa\_open—Open a Resource Manager

A transaction manager (TM) calls the `xa_open()` function to initialize a resource manager (RM) and prepare it for use in a distributed transaction processing environment. It must be called before any other XA function calls are made.

The transaction manager will issue an `xa_open` function to each resource manager that is defined in the TPF system at restart time.

### Format

```
#include (i$tmcr.h)
int      xa_open(char *xa_info, int rmid, long flags);
```

#### **xa\_info**

A pointer to a null-terminated character string that contains specific information for the resource manager. The maximum length of this string is 256 bytes (including the null terminator). **xa\_info** may point to an empty string if the resource manager does not require specific information to be available. **xa\_info** must not be a null pointer.

#### **rmid**

An integer assigned by the transaction manager that can be used to uniquely identify the called resource manager. The TM passes the **rmid** on subsequent calls to XA routines to identify the resource manager. This identity remains constant until the TM closes the resource manager.

#### **flags**

The only value supported is `TMNOFLAGS`.

### Normal Return

<b>XA_OK</b>	Normal completion.
--------------	--------------------

### Error Return

<b>XAER_RMERR</b>	An error occurred when opening the resource.
<b>XAER_INVAL</b>	Incorrect arguments were specified.

### Programming Considerations

- You must have authorization to issue a restricted C function.
- From the perspective of the resource manager, pointer **xa\_info** is valid only for the duration of the call to the `xa_open()` function; that is, once the function ends, the transaction manager is allowed to invalidate where **xa\_info** points. Resource managers are encouraged to use private copies of **\*xa\_info** after the function ends.

### Examples

```
#include (i$tmcr.h)

if (xa_open(char *xa_info, int rmid, long flags) != XA_OK)
{
    /* handle error condition */
    :
}
else {
    /* continue with normal processing */
}
```

```
⋮  
}
```

## Related Information

“xa\_end—End Work Performed for a Transaction Branch” on page 710.

## xa\_prepare—Prepare to Commit

The transaction manager (TM) calls the `xa_prepare` function to request a resource manager (RM) to prepare for commitment any work performed on for the transaction branch. The resource manager places any resources that are held or modified in such a state that it can make the results permanent when it receives a commit request.

Once this function successfully returns, the resource manager must guarantee that the transaction branch can be either committed or rolled back regardless of failures. A resource manager cannot erase its knowledge of a branch until the transaction manager calls either the `xa_commit` or `xa_rollback` function to complete the branch.

## Format

```
#include (<xtmcr.h>)
int      xa_prepare(XID *xid, int rmid, long flags);
```

### **xid**

A pointer to an exchange identification (XID) structure. **xid** is a unique identifier assigned by the transaction manager for each transaction branch.

### **rmid**

An integer, assigned by the TM, that can be used to uniquely identify the called resource manager instance.

### **flags**

#### **TMNOFLAGS**

The only value supported.

## Normal Return

<b>XA_OK</b>	Normal completion.
<b>XA_RDONLY</b>	The transaction branch was read-only and has been committed.

## Error Return

<b>XA_RBROLLBACK</b>	The resource manager did not prepare to commit the work for the transaction branch because of an error condition. The resource manager has rolled back the branch's work and has released all held resources.
<b>XAER_RMERR</b>	An error occurred in preparing to commit the work of the transaction branch. The specified <b>xid</b> may or may not have been prepared.
<b>XAER_RMFAIL</b>	An error occurred that makes the resource manager unavailable. The specified <b>xid</b> may or may not have been prepared.
<b>XAER_NOTA</b>	The specified <b>xid</b> is not known by the resource manager.
<b>XAER_INVAL</b>	Incorrect arguments were specified.
<b>XAER_PROTO</b>	The routine was called incorrectly.



## Programming Considerations

- You must have authorization to issue a restricted C function.
- From the perspective of the resource manager, pointer **xid** is valid only for the duration of the call to the `xa_prepare()` function; that is, once the function ends, the transaction manager is permitted to invalidate where **xid** points. Resource managers are encouraged to use private copies of **\*xid** after the function ends.

## Examples

```
#include (i$tmcr.h)

    if (xa_prepare(XID *xid, int rmid, long flags) != XA_OK)
    {
        /* handle error condition */
        :
    }
    else {
        /* continue with normal processing */
        :
    }
```

## Related Information

- “xa\_start—Start Work for a Transaction Branch” on page 720
- “xa\_open—Open a Resource Manager” on page 712.

## xa\_recover—Get a List of Prepared Transaction Branches

The transaction manager (TM) calls the `xa_recover` function during recovery to obtain a list of transaction branches that are currently in a prepared state.

The caller points **xids** to an array into which the resource manager (RM) places exchange identifications (XIDs) for those transactions and sets the count to the maximum number of XIDs that fit into the array. One or more `xa_recover` calls may be used in a recovery scan. The **flags** parameter defines when a recovery scan should start or end, or start and end. The start of a recovery scan moves the cursor to the start of a list of prepared transactions. Throughout the recovery scan, the cursor marks the current position in the list. Each call advances the cursor past the set of XIDs it returns.

If successful, the `xa_recover` function places zero or more XIDs in the space pointed to by **xids**. The function returns the number of XIDs it has placed there. If this value is less than **count**, there are no more XIDs to recover and the current scan ends; that is, the transaction manager does not need to call the `xa_recover` with the `TMENDSCAN` flag.

It is the responsibility of the transaction manager to ignore XIDs that do not belong to it.

## Format

```
#include (i$tmcr.h)
int      xa_recover(XID *xids, long count, int rmid, long flags);
```

### **xids**

A pointer to an array of exchange identification (XID) structures. The size of the array is described by the **count** parameter. **xids** is the area where the resource manager returns the XIDs that are in the prepared state.

### **count**

The maximum number of XIDs that can be stored in **xids**.

### **rmid**

An integer, assigned by the transaction manager, that can be used to uniquely identify the called resource manager instance.

### **flags**

#### **TMSTARTSCAN**

This flag indicates that the `xa_recover()` function should start a recover scan for the thread of control and position the cursor to the start of the list. XIDs are returned from that point. If a recovery scan is already open, the same result occurs as if the recovery scan ended and then restarted.

#### **TMENDSCAN**

This flag indicates that the `xa_recover()` function should end the recover scan after returning the XIDs. If the flag is used with `TMSTARTSCAN`, the single `xa_recover()` function call starts and ends a scan.

#### **TMNOFLAGS**

This flag must be used when no other flags are set. A recovery scan must already be started. XIDs are returned at the current cursor position.

## Normal Return

**>=0**

The total number of XIDs it returned in **xids**.

## Error Return

<b>XAER_RMERR</b>	An error occurred while determining the XIDs to return.
<b>XAER_RMFAIL</b>	An error occurred that makes the resource manager unavailable. The specified XID may or may not have been prepared.
<b>XAER_INVAL</b>	The pointer <b>xids</b> is null and <b>count</b> is greater than zero, <b>count</b> is negative, an incorrect flag was specified, or the ECB does not have a recovery scan open and did not specify TMSTARTSCAN in <b>flags</b> .
<b>XAER_PROTO</b>	The routine was called incorrectly.

## Programming Considerations

- You must have authorization to issue a restricted C function.
- If **xids** points to a buffer that cannot hold all of the XIDs requested, the `xa_recover` function may overwrite the caller's data space.

## Examples

```
#include (i$tmcr.h)

    if (xa_recover(XID *xids, long count, int rmid, long flags) < 0)
    {
        /* handle error condition */
        :
    }
    else {
        /* continue with normal processing */
        :
    }
```

## Related Information

- “xa\_commit—Commit Work Done for a Transaction Branch” on page 708
- “xa\_open—Open a Resource Manager” on page 712
- “xa\_rollback—Roll Back Work Done for a Transaction Branch” on page 718.

## xa\_rollback—Roll Back Work Done for a Transaction Branch

The transaction manager (TM) calls the `xa_rollback` function to roll back the work performed at a resource manager for the transaction branch. Any resources held by the resource manager (RM) for the branch are released, and those that have been modified are restored to their value at the start of the branch.

### Format

```
#include (i$tmcr.h)
int      xa_rollback(XID *xid, int rmid, long flags);
```

#### **xid**

A pointer to an exchange identification (XID) structure. **xid** is a unique identifier assigned by the TM for each transaction branch.

#### **rmid**

An integer, assigned by the TM, that can be used to uniquely identify the called RM instance.

#### **flags**

##### **TMNOFLAGS**

The only value supported.

### Normal Return

<b>XA_OK</b>	Normal completion.
--------------	--------------------

### Error Return

<b>XA_RBROLLBACK</b>	The resource manager has rolled back the work of the transaction branch and has released all held resources. This value is returned when the branch was already marked rollback-only.
<b>XAER_RMERR</b>	An error occurred when rolling back the transaction branch. The transaction branch is corrupted when returning this error.
<b>XAER_RMFAIL</b>	An error occurred that makes the resource manager unavailable. The specified <b>xid</b> may or may not have been prepared.
<b>XAER_NOTA</b>	The specified <b>xid</b> is not known by the resource manager.
<b>XAER_INVAL</b>	Incorrect arguments were specified.
<b>XAER_PROTO</b>	The routine was called incorrectly.

### Programming Considerations

- You must have authorization to issue a restricted C function.
- From the perspective of the resource manager, the pointer **xid** is valid only for the duration of the call to the `xa_rollback()` function; that is, once the function ends, the transaction manager is permitted to invalidate where **xid** points. Resource managers are encouraged to use private copies of **\*xid** after the function ends.

## Examples

```
#include (<xtmcr.h>)

if (xa_rollback(XID *xid, int rmid, long flags) != XA_OK)
{
    /* handle error condition */
    :
}
else {
    /* continue with normal processing */
    :
}
```

## Related Information

- “xa\_commit—Commit Work Done for a Transaction Branch” on page 708
- “xa\_open—Open a Resource Manager” on page 712
- “xa\_prepare—Prepare to Commit” on page 714.

## xa\_start–Start Work for a Transaction Branch

The transaction manager (TM) calls the `xa_start` function to inform a resource manager (RM) that an application may do work for a transaction branch.

The `tx_resume_tpf` function must be entered from the entry control block (ECB) that suspended the transaction branch.

### Format

```
#include (i$tmcr.h)
int      xa_start(XID *xid, int rmid, long flags);
```

#### **xid**

A pointer to an exchange identification (XID) structure. **xid** is a unique identifier assigned by the TM for each transaction branch.

#### **rmid**

An integer, assigned by the TM, that can be used to uniquely identify the called resource manager instance.

#### **flags**

##### **TMRESUME**

Used by the `tx_resume_tpf` function.

##### **TMNOFLAGS**

Used by the `tx_begin` function.

### Normal Return

<b>XA_OK</b>	<code>xa_start</code> normal completion.
--------------	--

### Error Return

<b>XAER_RMERR</b>	An error occurred when associating the transaction branch with the thread of control.
<b>XAER_RMFAIL</b>	An error occurred that makes the resource manager unavailable.
<b>XAER_DUPID</b>	The XID already exists in the resource manager and the TMRESUME flag is not set.
<b>XAER_OUTSIDE</b>	The resource manager is doing work outside any global transaction for the application.
<b>XAER_NOTA</b>	The TMRESUME flag is set and the specified <b>xid</b> is not known by the resource manager.
<b>XAER_INVAL</b>	Incorrect arguments were specified.
<b>XAER_PROTO</b>	The routine was called incorrectly.
<b>XAER_RBROLLBACK</b>	A resource manager has not associated the transaction branch with the thread of control and has marked the transaction branch rollback-only.

### Programming Considerations

- You must have authorization to issue a restricted C function.
- From the perspective of the resource manager, the pointer **xid** is valid only for the duration of the call to the `xa_start()` function; that is, once the function ends,

the transaction manager is permitted to invalidate where **xid** points. Resource managers are encouraged to use private copies of **\*xid** after the function ends.

## Examples

```
#include (i$tmcr.h)

    if (xa_start(XID *xid, int rmid, long flags) != XA_OK)
    {
        /* handle error condition */
        :
    }
    else {
        /* continue with normal processing */
        :
    }
```

## Related Information

- “xa\_end—End Work Performed for a Transaction Branch” on page 710
- “xa\_open—Open a Resource Manager” on page 712.

**xa\_start**



---

## TPF/APPC Basic Conversation Functions

The function provided by the TPF Advanced Program-to-Program Communications (TPF/APPC) interface is an implementation of the IBM Advanced Program-to-Program Communications (APPC) architecture. TPF/APPC is an interface that allows TPF transaction programs to communicate with remote SNA nodes that have implemented the APPC interface using LU 6.2 protocols. See the *TPF ACF/SNA Data Communications Reference* for more information about TPF/APPC support. Before using this support, you must be familiar with the SNA LU 6.2 protocol architecture as defined in the following publications:

- *IBM Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2*
- *IBM Systems Network Architecture LU 6.2 Reference: Peer Protocols*
- *IBM Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2.*

The TPF/APPC functions provide the interface for all TPF system-supported verbs defined in the LU 6.2 architecture. This chapter contains a description of the TPF/APPC basic conversation functions in their general form and an alphabetic listing of each function.

**Note:** The functions `tppc_get_type` and `tppc_wait` are defined by the LU 6.2 architecture as type-independent verbs; that is, they provide functions that span both basic conversations and mapped conversations. They are listed in this chapter because they are processed with the same interface as the basic conversation functions.

The following information is shown for each function:

**Format**            The function prototype and a description of any parameters.

**Description**      What service the function provides.

**Return Codes**    A table containing a list of return codes for the specific function.

**Note:** This section replaces the Normal Return and Error Return sections that are used in "TPF API Functions" on page 1.

### Programming Considerations

Remarks that help the programmer to understand the correct use of the function and any side effects that can occur when the function is executed. If the use of a particular function affects the use of another function, that is also described.

**Example**            A code segment showing a sample function call.

### Related Functions

Where to find additional information pertinent to this function.

**Note:** Remember to include `tpfeq.h` in the source program, even though it is not shown in any of the function prototypes. The `tpfeq.h` header must be the first header file included in any TPF C source module.

For descriptions of the TPF/APPC mapped conversation functions, see "TPF/APPC Mapped Conversation Functions" on page 793.

---

## General Format

Figure 1 shows the general form for the TPF/APPC functions.

```
tppc_verbname  
(parameter_1,  
    .  
    .  
    .  
    parameter_n);
```

Figure 1. General Form for TPF/APPC Functions

Each function begins with **tppc\_**, followed by the specific *verbnam*e, which specifies the function to be executed. This is followed by a set of positional parameters, which means that you must code each parameter in the order shown for the specific function. Parameter values indicate either an option or a pointer to the location of a field. You must specify the options exactly as shown in the function descriptions. Options are indicated by uppercase bold print, such as **OPTION**. The valid parameters for each TPF/APPC function depend on the *verbnam*e you specify.

Table 36 on page 725 is a summary of the supported verb names and their associated parameters. The table also contains:

- An indication of whether the parameter is passed to or returned from the TPF/APPC processing components
- A brief description of the parameter
- The equivalent parameter defined in the LU 6.2 architecture.

More detailed information for each parameter is provided in the individual function descriptions later in this chapter.

Table 36. TPF/APPC Basic Conversation Verbs and Valid Keywords

Verb Name	Positional Parameter	Passed or Returned	Description	Architecture Equivalent
activate_on_confirmation	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	verb	Passed	This specifies the architecture's verb function to be performed in conjunction with this verb. Valid choices are: • <b>AOC_CONFIRM</b> • <b>AOC_DEALLOC</b> • <b>AOC_P_T_R</b> .	None
	level	Passed	This specifies the data level that has a block to be passed to the activated program.	None
	field	Passed	This specifies a field that contains data to be passed to the activated program.	None
	pgm	Passed	This specifies the TPF E-type program to be activated when there is data or other information available to satisfy the CONFIRM request. The program specified must be defined in the TPF transaction program name table (TPNT) as TYPE=AOR.	None
<b>Note:</b> The tppc_activate_on_confirmation function is a TPF-only extension to the LU 6.2 architecture. There is no equivalent verb defined in the architecture.				
activate_on_receipt	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	level	Passed	This specifies the data level that contains a block to be passed to the activated program.	None
	field	Passed	This specifies a field that contains a data to be passed to the activated program.	None
	pgm	Passed	This specifies the TPF E-type program to be activated when there is data or other information available to satisfy the RECEIVE request. The program specified must be defined in the TPF TPNT as TYPE=AOR.	None
<b>Note:</b> The tppc_activate_on_receipt function is a TPF-only extension to the LU 6.2 architecture. There is no equivalent verb defined in the architecture.				
allocate	resid	Returned	This specifies where the resource ID is to be returned. The resource ID uniquely identifies this conversation from all others and is used when calling other TPF/APPC functions for this conversation. The resource ID is 4 bytes long.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	luname	Passed	This specifies the fully qualified name of the remote LU.	LU_NAME
	tpn	Passed	This specifies the name of the remote transaction program used on this conversation.	TPN

Table 36. TPF/APPC Basic Conversation Verbs and Valid Keywords (continued)

Verb Name	Positional Parameter	Passed or Returned	Description	Architecture Equivalent
	mode	Passed	This specifies the name used to designate the properties of the session to be allocated.	MODE_NAME
	sync	Passed	This specifies the synchronization level allowed on the allocated conversation. The values <b>ALLOCATE_SYNC_NONE</b> and <b>ALLOCATE_SYNC_CONFIRM</b> provide support for the LU 6.2 architecture's NONE and CONFIRM options, respectively. Only the <b>NONE</b> and <b>CONFIRM</b> options are supported. The TPF system does not support the architecture's SYNCPT option.	SYNC_LEVEL
	rcontrol	Passed	This specifies the condition on which control is returned to the issuing program. The value <b>ALLOCATE_RCONTROL_WSA</b> specifies to return control when a session is allocated for use by the conversation. The value <b>ALLOCATE_RCONTROL_IMM</b> specifies to return control immediately if no contention-winner session is currently active and available. The TPF system does not support the other options defined by the LU 6.2 architecture.	RETURN_CONTROL
	type	Passed	This specifies the type of conversation. The value <b>ALLOCATE_TYPE_BASIC</b> provides support for the BASIC_CONVERSATION option defined by the LU 6.2 architecture. The value <b>ALLOCATE_TYPE_MAPPED</b> provides support for the MAPPED_CONVERSATION option defined by the LU 6.2 architecture.	TYPE
	pip	Passed	This specifies whether or not program initialization parameters are supported. The value <b>ALLOCATE_PIP_NO</b> provides support for the NO option defined by the LU 6.2 architecture. The TPF system does not support the architecture's YES option.	PIP
	security	Passed	This specifies the security information used to verify the identity of the end users of the conversation. The value <b>ALLOCATE_SECURITY_NO</b> provides support for the NO option defined by the LU 6.2 architecture. This indicates that access security information is omitted on this conversation. The TPF system does not support the architecture's SAME and PGM options.	SECURITY
confirm	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	rtsrcvd	Returned	This specifies where an indication is to be returned that designates whether or not a REQUEST_TO_SEND was received on this conversation.	REQUEST_TO_SEND_RECEIVED
confirmed	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE

Table 36. TPF/APPC Basic Conversation Verbs and Valid Keywords (continued)

Verb Name	Positional Parameter	Passed or Returned	Description	Architecture Equivalent
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
deallocate	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	type	Passed	This specifies the type of deallocation to be done.	TYPE
	logdata	Passed	This specifies whether or not error information is to be logged. <b>DEALLOCATE_LOGDATA_NO</b> provides support for the NO option defined by the LU 6.2 architecture. The TPF system does not support the architecture's YES option.	LOG_DATA
flush	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
get_attributes	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	ownname	Returned	This specifies where the local LU name is returned.	OWN_FULLY_QUALIFIED_LU_NAME
	pluname	Returned	This specifies where the name of the partner LU is returned. The partner LU name and the fully qualified partner LU name are one and the same.	PARTNER_LU_NAME
	mode	Returned	This specifies where the mode name used for this conversation is returned.	MODE_NAME
	sync	Returned	This specifies where the synchronization level used for this conversation is returned. The values <b>GET_ATTRIBUTES_SYNC_NONE</b> and <b>GET_ATTRIBUTES_SYNC_CONFIRM</b> provide support for the NONE and CONFIRM options defined by the LU 6.2 architecture. The TPF system does not support the architecture's SYNCPT option.	SYNCH_LEVEL
get_type	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	type	Returned	This specifies where the conversation type is returned.	TYPE
post_on_receipt	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE

Table 36. TPF/APPC Basic Conversation Verbs and Valid Keywords (continued)

Verb Name	Positional Parameter	Passed or Returned	Description	Architecture Equivalent
	fill	Passed	This specifies when posting should occur based on the logical record format. The value <b>POST_ON_RECEIPT_FILL_LL</b> provides support for the architecture's LL option. This specifies that posting occurs when a complete or truncated logical record is received, or when a part of a logical record is received that is at least equal in length to that specified on the <b>length</b> parameter. The TPF system does not support the BUFFER option defined by the LU 6.2 architecture.	FILL
	length	Passed	This specifies the minimum amount of data that the LU must receive before the conversation can be posted. This parameter is used along with the <b>fill</b> parameter to determine when to post the conversation.	LENGTH
prepare_to_ receive	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	type	Passed	This specifies the type of PREPARE_TO_RECEIVE to be performed. The values <b>PREP_TO_RECEIVE_TYPE_FLUSH</b> , <b>PREP_TO_RECEIVE_TYPE_CONFIRM</b> , and <b>PREP_TO_RECEIVE_TYPE_SYNC</b> provide support for the architecture's FLUSH, CONFIRM, and SYNC_LEVEL options, respectively. If <b>FLUSH</b> or <b>CONFIRM</b> is specified, the appropriate verb function is performed. If <b>SYNC</b> is specified, the synchronization level specified on the ALLOCATE is used to determine which verb function is to be performed. If the synchronization level was <b>NONE</b> , the FLUSH verb function is performed. If the synchronization level was <b>CONFIRM</b> , the CONFIRM verb function is performed.	TYPE
	locks	Passed	This specifies when control is to be returned, when the CONFIRM verb is used or implied on the <b>type</b> parameter. The value <b>PREP_TO_RECEIVE_LOCKS_SHORT</b> provides support for the SHORT option defined by the LU 6.2 architecture. This causes control to be returned upon the receipt of a positive reply to CONFIRM. The TPF system does not support the architecture's LONG option.	LOCKS
receive	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	whatrcv	Returned	This specifies where the WHAT_RECEIVED indication is returned. The WHAT_RECEIVED indication may indicate data, confirmation, or conversation status.	WHAT_RECEIVED

Table 36. TPF/APPC Basic Conversation Verbs and Valid Keywords (continued)

Verb Name	Positional Parameter	Passed or Returned	Description	Architecture Equivalent
	rtsrcvd	Returned	This specifies where the REQUEST_TO_SEND indication is to be returned.	REQUEST_TO_SEND_ RECEIVED
	fill	Passed	This specifies when the receive should be satisfied based on the logical record format. The value <b>RECEIVE_FILL_LL</b> provides support for the architecture's LL option. This specifies that posting occurs when a complete or truncated logical record is received, or when a part of a logical record is received that is at least equal in length to that specified on the <b>length</b> parameter. The TPF system does not support the BUFFER option defined by the LU 6.2 architecture.	FILL
	wait	Passed	This specifies that the RECEIVE_AND_WAIT verb function should be performed. The value <b>RECEIVE_WAIT_YES</b> specifies the RECEIVE_AND_WAIT verb. The TPF system does not support the RECEIVE_IMMEDIATE verb defined by the LU 6.2 architecture.	
	data	Passed	This specifies the location in memory that is to receive the data.	DATA
	length	Passed and Returned	This specifies the maximum length of data that the TP can receive. When control is returned to the TP, this variable contains the actual amount of data the program received up to the maximum. If the program receives information other than data, this variable remains unchanged.	LENGTH
request_to_ send	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
send_data	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	rtsrcvd	Returned	This specifies where the REQUEST_TO_SEND indication is to be returned.	
	data	Passed	This specifies the location in memory that contains the data to be sent.	DATA
	length	Passed	This specifies the length of the data to be sent. This data length is in no way related to the length of a logical record. It is used only to determine the length of the data located at the address specified on the DATA parameter.	LENGTH
send_error	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE

Table 36. TPF/APPC Basic Conversation Verbs and Valid Keywords (continued)

Verb Name	Positional Parameter	Passed or Returned	Description	Architecture Equivalent
	rtsrcvd	Returned	This specifies where the REQUEST_TO_SEND indication is to be returned.	REQUEST_TO_SEND_ RECEIVED
	type	Passed	This specifies the type of error that was detected. The value <b>SEND_ERROR_TYPE_PROG</b> specifies that the transaction program detected an error. The value <b>SEND_ERROR_TYPE_SVC</b> specifies that the TPF service program detected an error.	TYPE
	logdata	Passed	This specifies whether or not error information is to be logged. The value <b>SEND_ERROR_LOGDATA_NO</b> provides support for the NO option defined by the LU 6.2 architecture. The TPF system does not support the architecture's YES option.	LOG_DATA
test	resid	Passed	This specifies the resource ID returned by the ALLOCATE verb or the resource ID assigned by an incoming ATTACH.	RESOURCE
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	test	Passed	This specifies which condition is to be tested. The value <b>TEST_TEST_POSTED</b> specifies to test whether the conversation was posted. The value <b>TEST_TEST_RTSCVD</b> specifies to test whether a REQUEST_TO_SEND was received.	TEST
wait	residl	Passed	This specifies where a list of resource IDs are located.	RESOURCE_LIST
	rcode	Returned	This specifies where the return code is to be returned.	RETURN_CODE
	respstd	Returned	This specifies where the resource ID of the conversation posted is returned.	RESOURCE_POSTED
<b>Note:</b> <b>Passed</b> indicates that the value or option is passed from the transaction program to the verb processing component. <b>Returned</b> indicates that the value or option is returned to the transaction program from the verb processing component.				



## Return Codes for Basic Conversation Functions

Each TPF/APPC function returns a 6-byte return code. The return code consists of a 2-byte primary return code placed in an unsigned short integer and a 4-byte secondary return code placed in an unsigned long integer.

The primary return code contains a zero value for normal return. In general, when the primary return code is zero, the secondary return code has no meaning. The following functions are exceptions to this; for these functions the secondary return code does contain meaningful information on a normal return:

```
tppc_activate_on_confirmation
tppc_activate_on_receipt
tppc_test
tppc_wait.
```

The primary return code contains a nonzero value for an error return. In general, the secondary return code provides more detailed error information. There are exceptions to this; some of the primary return codes do not have a secondary code associated with them.

Table 37 contains a list of all the primary return codes and their meanings, and Table 38 on page 732 contains a list of all the secondary return codes and their meanings. Not all return codes are possible for every function. The individual TPF/APPC function descriptions document the valid return codes for that TPF/APPC function. The symbolic names of the return code values are defined in the `tppc.h` include file.

Table 37. Primary Return Codes

Symbolic Name	Hex Value	Meaning
LU62RC_OK	0000	The function completed successfully.
LU62RC_PARAMETER_CHECK	0001	An invalid parameter was detected. Check the secondary return code.
LU62RC_STATE_CHECK	0002	An invalid state was detected. Check the secondary return code.
LU62RC_ALLOC_ERROR	0003	An allocation error occurred. Check the secondary return code.
LU62RC_DLLOC_ABEND_PGM	0006	An error occurred, and the conversation was deallocated.
LU62RC_DLLOC_ABEND_SVC	0007	An error occurred, and the conversation was deallocated.
LU62RC_DLLOC_ABEND_TMR	0008	An error occurred, and the conversation was deallocated.
LU62RC_DLLOC_NORMAL	0009	The conversation was deallocated.
LU62RC_POSTING_NOTACTIV	000B	Posting is not active for the specified conversation.
LU62RC_PGMERR_NOTRUNC	000C	A program error occurred. Data was not truncated.
LU62RC_PGMERR_TRUNC	000D	A program error occurred. Data was truncated.
LU62RC_PGMERR_PURGING	000E	A program error occurred. Data was purged.
LU62RC_CONVFAIL_RETRY	000F	The conversation failed due to a session outage.
LU62RC_CONVFAIL_NORETRY	0010	The conversation failed due to a protocol error.

Table 37. Primary Return Codes (continued)

Symbolic Name	Hex Value	Meaning
LU62RC_SVCERR_NOTRUNC	0011	A service program error occurred. Data was not truncated.
LU62RC_SVCERR_TRUNC	0012	A service program error occurred. Data was truncated.
LU62RC_SVCERR_PURGING	0013	A service program error occurred. Data was purged.
LU62RC_LLRCV_UNSUCESSFUL	0014	The condition checked by the <code>tppc_test</code> function was not set. See the <code>tppc_test</code> function description for further information.
LU62RC_ALLOC_UNSUCESSFUL	0015	The <code>tppc_allocate</code> function with <b>ALLOCATE_RCONTROL_IMM</b> was not able to allocate a session to this conversation immediately.
LU62RC_TPF_ABEND	FFFF	This indicates a system abend or time-out.

Table 38. Secondary Return Codes

Symbolic Name	Hex Value	Meaning
LU62RC_POSTED_DATA	00000000	The conversation was posted. The information available is data.
LU62RC_POSTED_NOTDATA	00000001	The conversation was posted. The information available is <i>not</i> data.
LU62RC_PK_BAD_TCBID	00000001	A parameter check was detected. The TCB ID is invalid.
LU62RC_PK_BAD_CONVID	00000002	A parameter check was detected. The resource ID is invalid.
LU62RC_PK_EMPTY_RESIDL	00000003	A parameter check was detected. The resource list specified on the <code>tppc_wait</code> function is empty.
LU62RC_ALLOCERR_NORETRY	00000004	An allocation error occurred. Do not retry the request.
LU62RC_ALLOCERR_RETRY	00000005	An allocation error occurred. Retry the request.
LU62RC_INVALID_LENGTH	00000006	A parameter check was detected. The <b>length</b> parameter coded on the <code>tppc_post_on_receipt</code> , <code>tppc_receive</code> , or <code>tppc_send_data</code> function exceeds the maximum value of 32,767.
LU62RC_AOC_INCOMPLETE	00000009	The <code>tppc_activate_on_confirmation</code> function was accepted. The program specified ( <b>pgm</b> ) is activated when the confirm request is complete.
LU62RC_AOR_INCOMPLETE	00000009	The <code>tppc_activate_on_receipt</code> function verb was accepted. The program specified ( <b>pgm</b> ) is activated after the data or other information is received.
LU62RC_SECURITY_NOT_VALID	080F6051	An allocation error occurred. The security information is invalid.
LU62RC_SKCNFRM_BADSTATE	00000032	A state check occurred. This can occur if <code>tppc_confirm</code> or <code>tppc_activate_on_confirmation</code> with <b>AOC_CONFIRM</b> is called when the conversation is not in <code>send</code> state or while in the middle of sending a logical record.

Table 38. Secondary Return Codes (continued)

Symbolic Name	Hex Value	Meaning
LU62RC_SKCNFRM_INVALID	00000033	A state check occurred. tppc_confirm or tppc_activate_on_confirmation with <b>AOC_CONFIRM</b> was called for a conversation allocated with <b>ALLOCATE_SYNC_NONE</b> .
LU62RC_SKCNFMD_BADSTATE	00000041	A state check occurred. tppc_confirmed was called, but the conversation is not in <u>received-confirm</u> , <u>received-confirm-send</u> , or <u>received-confirm-deallocate</u> state.
LU62RC_PKDLLOC_BADTYPE	00000051	A parameter check was detected. tppc_activate_on_confirmation with <b>AOC_DEALLOC</b> or tppc_deallocate with <b>DEALLOCATE_TYPE_CONFIRM</b> was requested on a conversation allocated with <b>ALLOCATE_SYNC_NONE</b> .
LU62RC_SKDLLOC_FLUSH	00000052	A state check occurred. This can occur if tppc_deallocate is called with <b>DEALLOCATE_TYPE_FLUSH</b> when the conversation is not in <u>send</u> state or while in the middle of sending a logical record.
LU62RC_SKDLLOC_CONFIRM	00000053	A state check occurred. This can occur if tppc_deallocate with <b>DEALLOCATE_TYPE_CONFIRM</b> or tppc_activate_on_confirmation with <b>AOC_DEALLOC</b> is called when the conversation is not in <u>send</u> state or while in the middle of sending a logical record.
LU62RC_SKDLLOC_ABEND	00000056	A state check occurred. tppc_deallocate was called with <b>DEALLOCATE_TYPE_ABEND</b> , but the conversation is in <u>end-conversation</u> state.
LU62RC_SKDLLOC_LOCAL	00000057	A state check occurred. tppc_deallocate was called with <b>DEALLOCATE_TYPE_CONFIRM</b> , but the conversation is not in <u>end-conversation</u> state.
LU62RC_SKFLUSH_BADSTATE	00000061	A state check occurred. tppc_flush was called, but the conversation is not in <u>send</u> state.
LU62RC_SKPOSTR_BADSTATE	00000092	A state check occurred. tppc_post_on_receipt was called, but the conversation is not in <u>receive</u> state.
LU62RC_PKPTRCV_INVTYPE	000000A1	A parameter check was detected. tppc_activate_on_confirmation with <b>AOC_P_T_R</b> or tppc_prepare_to_receive with <b>PREP_TO_RECEIVE_TYPE_CONFIRM</b> was requested on a conversation allocated with <b>ALLOCATE_SYNC_NONE</b> .
LU62RC_SKPTRCV_BADSTATE	000000A3	A state check occurred. This can occur if tppc_prepare_to_receive is called when the conversation is not in <u>send</u> state or while in the middle of sending a logical record.
LU62RC_SKRECEV_BADSTATE	000000B1	A state check occurred. This can occur if tppc_receive or tppc_activate_on_receipt is called when the conversation is not in <u>send</u> state or <u>receive</u> state, or while in the middle of sending a logical record.

Table 38. Secondary Return Codes (continued)

Symbolic Name	Hex Value	Meaning
LU62RC_SKRTSND_BADSTATE	000000E1	A state check occurred. <code>tppc_request_to_send</code> was called, but the conversation is not in <u>send</u> , <u>receive</u> , <u>received-confirm</u> , <u>received-confirm-send</u> , or <u>received-confirm-deallocate</u> state.
LU62RC_PKSENDD_BADLL	000000F1	A parameter check was detected. This can occur if a <code>tppc_send_data</code> is called without a data message block, with an invalid logical length, or if there is bad file chaining in the data block.
LU62RC_PK_BAD_PROG	000000F2	A parameter check was detected. A <code>tppc_activate_on_confirmation</code> or <code>tppc_activate_on_receipt</code> function was called, but the program specified on the <b>pgm</b> parameter is not defined in the TPNT.
LU62RC_SKSENDD_BADSTATE	000000F2	A state check occurred. <code>tppc_send_data</code> was called, but the conversation is not in <u>send</u> state.
LU62RC_INVALID_MODE_NAME	000000F3	An invalid mode name has been passed on the <code>tppc_allocate</code> call. This error occurs when the mode name specified is <b>SNASVCMG</b> . This mode name is reserved for CNOS (change number of sessions) processing.
LU62RC_PK_BAD_PARM	000000F3	A parameter check was detected. A <code>tppc_activate_on_confirmation</code> or <code>tppc_activate_on_receipt</code> function was called, but there is no block on the data level specified by the <b>level</b> parameter.
LU62RC_SKSENDE_BADSTATE	00000111	A state check occurred. <code>tppc_send_error</code> was called, but the conversation is in an invalid state.
LU62RC_NOT_RCV_STATE	00000122	A state check occurred. This can occur if <code>tppc_test</code> with <b>test=TEST_TEST_POSTED</b> or <code>tppc_wait</code> is called, but the conversation is not in <u>receive</u> state. This error also occurs when <code>tppc_test</code> with <b>test=TEST_TEST_RTSCVD</b> is called while the conversation is not in either <u>send</u> or <u>receive</u> state.
LU62RC_PK_BAD_OPTION	00C62074	An invalid parameter option was passed to a TPF/APPC C runtime library function. An OPR-C62074 dump is issued. Control returns immediately to the calling function.
LU62RC_TP_NOT_AVAIL_RETRY	084B6031	An allocation error occurred. The transaction program could not be started. One possible cause is the lack of resources. This is a temporary condition; retry the request.
LU62RC_TP_NOT_AVAIL_NO_RETRY	084C0000	An allocation error occurred. The transaction program could not be started. One possible cause is the lack of resources. This is a permanent condition; do not retry the request.
LU62RC_TPN_NOT_RECOGNIZED	10086021	An allocation error occurred. The transaction program name specified on the <code>tppc_allocate</code> call was not recognized by the remote LU.

Table 38. Secondary Return Codes (continued)

Symbolic Name	Hex Value	Meaning
LU62RC_PIP_NOT_SPECIFIED_CORRECTLY	10086032	An allocation error occurred. The remote LU rejected the allocation request because the PIP parameter specified did not agree with the PIP value defined by the remote program.
LU62RC_CONV_TYPE_MISMATCH	10086034	An allocation error occurred. This error indicates that the remote LU rejected the allocation request because the remote transaction program does not support the conversation type requested.
LU62RC_SYNLVL_NOTSUPPORT	10086041	An allocation error occurred. tppc_allocate was called with a value for the <b>sync</b> argument that the remote LU does not support.

## Programming Considerations for Basic Conversation Functions

- You must code all parameters described for each function in the order shown.
- The conversation must be in a valid state for the particular *verbname* specified. The valid states for each function are listed in the “Programming Considerations” section for that function. See *TPF ACF/SNA Data Communications Reference* for more information about the finite state machine and valid states.
- You can execute these functions on any I-stream.
- The TPF/APPC function generates a `crosc_entrc` to the TPF/APPC verb router program, a real-time ECB-controlled program. Additional E-type programs are activated based on the *verbname*, the passed parameters, and the status and conditions that may be present for the conversation. There must be at least 7 program nesting levels available.
- The TPF/APPC support programs save and restore the contents of EBW016 – EBW103 and the EBX area of the calling ECB.
- The TPF/APPC support programs use various data levels of the ECB to perform their processing. If the data levels used by the support programs have blocks attached when the TPF/APPC function is executed, the support programs detach the blocks from the ECB (using DETAC) and attach them back to the ECB (using ATTAC) before returning control to the calling program. Data level 15 (DF) is always used. Data levels 12–14 (DC–DE) are used at various times. The application can reduce the overhead inherent in the DETAC/ATTAC processing by insuring that these data levels are available when the TPF/APPC function is executed.
- If you code variables that contain enumeration types instead of coding the enumeration types directly, the expansion of the function creates more instructions, takes more space, and takes longer to execute.
- When a remote LU 6.2 transaction program starts a conversation with a TPF transaction program, the TPF system receives an ATTACH FMH5 record. See *TPF ACF/SNA Data Communications Reference* for a description of the ATTACH interface.
- The 2 ACTIVATE functions (`tppc_activate_on_confirmation` and `tppc_activate_on_receipt`) are TPF-only extensions to the LU 6.2 architecture. They allow you to implement a TPF transaction program with an asynchronous wait mechanism. After issuing either of the ACTIVATE verbs, the ECB (which represents the transaction program instance) is expected to call the TPF exit function. When the wait implied with the synchronous type verb (`tppc_receive`,

tppc\_confirm, tppc\_prepare\_to\_receive, tppc\_deallocate) completes, TPF/APPC support re-creates the TPF transaction program instance by activating the specified program with a new ECB.

- The TPF/APPC support always returns to the calling transaction program with a protection key of 1.
- See the individual function descriptions for programming considerations relevant to each supported verb.

## tppc\_activate\_on\_confirmation—Activate a Program after Confirmation Received

This function is an extension to the LU 6.2 architecture. This function allows the issuing TPF ECB to exit after sending out a `tppc_confirm`, `tppc_deallocate` with **DEALLOCATE\_TYPE\_CONFIRM** or `tppc_prepare_to_receive` with **PREP\_TO\_RECEIVE\_TYPE\_CONFIRM** request. TPF/APPC then activates a different ECB at the program specified when the reply to the confirm request has arrived and has been processed.

### Format

```
#include <tppc.h>
void tppc_activate_on_confirmation(unsigned int *resid,
                                  struct tppc_return_codes *rcode,
                                  enum t_aoc_verb verb,
                                  enum t_aoc_d1 level,
                                  unsigned char *field,
                                  unsigned char *pgm);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial **ALLOCATE** for this conversation or one that was assigned by an incoming **ATTACH**.

#### rcode

This is a pointer to the structure `tppc_return_codes`, defined in `tppc.h`, where the return code will be placed.

#### verb

This specifies the verb function that is to be executed. This argument must belong to the enumeration type `t_aoc_verb`, defined in `tppc.h`. Use one of the following values:

##### AOC\_CONFIRM

This value specifies that the `tppc_confirm` function should be executed.

##### AOC\_DEALLOC

This value specifies that the `tppc_deallocate` function should be executed with **type=DEALLOCATE\_TYPE\_CONFIRM**.

##### AOC\_P\_T\_R

This value specifies that the `tppc_prepare_to_receive` function should be executed with **type=PREP\_TO\_RECEIVE\_TYPE\_CONFIRM**.

#### level

This specifies a data level that contains a working storage block to be passed to the activated program. After the reply to the **CONFIRM** request arrives and is processed, the contents of the block on the data level specified is passed from the ECB issuing this verb to a 4K block on the same data level of the ECB created. This argument must belong to the enumeration type `t_aoc_d1`, defined in `tppc.h`. Use a value in the range of **AOC\_D0** to **AOC\_DA**. If no block is to be passed, use the value **AOC\_NO\_DL**.

#### field

This is a pointer to an 8-byte field that is passed to the program specified by **pgm** when it is activated. The data in this field is copied to **EBX000** of the new ECB that is created when the reply to the **CONFIRM** request has arrived and has been processed. If the **level** parameter is used instead, this parameter must have a value of **NULL**.



## tppc\_activate\_on\_confirmation

### pgm

This is a pointer to a 4-byte field that contains the name of the TPF real-time program to be activated after the reply to the CONFIRM request has arrived and has been processed. This TPF real-time segment must be defined in the TPF transaction program name table (TPNT) as an ACTIVATE\_ON\_RECEIPT target program. This is done with the ITPNT macro by specifying TYPE=AOR and by specifying the segment name as both the TPN name and the PGM name. See *TPF General Macros* for more information about the ITPNT macro.

## Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the tppc\_activate\_on\_confirmation function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_AOC_INCOMPLETE	....	00000009
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_PKDLLOC_BADTYPE	....	00000051
LU62RC_PKPTRCV_INVTYPE	....	000000A1
LU62RC_PK_BAD_PROG	....	000000F2
LU62RC_PK_BAD_PARM	....	000000F3
LU62RC_PK_BAD_OPTION	....	00C62074
LU62RC_STATE_CHECK	0002	
LU62RC_SKCNFRM_BADSTATE	....	00000032
LU62RC_SKCNFRM_INVALID	....	00000033
LU62RC_SKDLLOC_CONFIRM	....	00000053
LU62RC_SKPTRCV_BADSTATE	....	000000A3
LU62RC_TPF_ABEND	FFFF	

The following table contains a list of the primary and secondary return codes that can be passed to the program specified by **pgm** when it is activated. The return codes are based on the reply to the CONFIRM request.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_ALLOC_ERROR	0003	
LU62RC_TP_NOT_AVAIL_RETRY	....	084B6031
LU62RC_TP_NOT_AVAIL_NO_RETRY	....	084C0000
LU62RC_TPN_NOT_RECOGNIZED	....	10086021
LU62RC_PIP_NOT_SPECIFIED_CORRECTLY	....	10086032
LU62RC_CONV_TYPE_MISMATCH	....	10086034
LU62RC_SYNLVL_NOTSUPPORT	....	10086041



Symbolic Name	Primary Code	Secondary Code
LU62RC_DLLOC_ABEND_PGM	0006	
LU62RC_DLLOC_ABEND_SVC	0007	
LU62RC_DLLOC_ABEND_TMR	0008	
LU62RC_PGMERR_PURGING	000E	
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NO_RETRY	0010	
LU62RC_SVCERR_PURGING	0013	
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The conversation must be in send state.
- The value supplied in **resid** must be the resource ID returned by the `tppc_allocate` function or one that was assigned by an incoming ATTACH.
- You can call this function instead of `tppc_confirm`, `tppc_deallocate` with **DEALLOCATE\_TYPE\_CONFIRM** or `tppc_prepare_to_receive` with **PREP\_TO\_RECEIVE\_TYPE\_CONFIRM**. After the reply to the confirmation request arrives and is processed, TPF/APPC support activates the program specified by **pgm**. The return codes are passed to the new ECB in contiguous memory starting at EBX008. An ECB is not suspended while waiting for the reply to the CONFIRM request.

**Note:** The structure of the `tppc_return_codes` requires 2 alignment bytes between the primary and secondary return codes. This structure cannot be used to overlay the area starting at EBX008.

- If the return code is LU62RC\_OK, the ECB is no longer connected to the transaction program or the conversation. The ECB cannot issue any more TPF/APPC verbs.
- If the return code is any value other than LU62RC\_OK, the ECB is still connected to the transaction program and the conversation. However, because all other return codes indicate a serious error, this also indicates that a DEALLOCATE ABEND occurred for the conversation. The ECB cannot issue any more TPF/APPC verbs for this conversation but can issue TPF/APPC verbs for other conversations associated with the transaction program instance.

**Note:** The transaction program is identified with the transaction program identifier (TCB ID), and the conversation is identified with the conversation control block identifier (CCB ID). Both identifiers are defined in the ECB as EBTCBID and EBCCBID respectively.

- You must specify either the **level** parameter or the **field** parameter, but not both.
- Any working storage block passed on a data level to the activated program is released by TPF/APPC.
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

```
#include <tppc.h>

unsigned int          resource_id;
```

## tppc\_activate\_on\_confirmation

```
        struct tppc_return_codes    return_code;
        unsigned char               pgm[4];
        .
        .
        .
        .
tppc_activate_on_confirmation(&resource_id,&return_code,          \
                             AOC_CONFIRM,AOC_D5,NULL,pgm);      \
        .               /* Normal processing path                */
        .
        .
```

## Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_allocate—Allocate a Conversation” on page 745
- “tppc\_confirm—Send a Confirmation Request” on page 750
- “tppc\_deallocate—Deallocate a Conversation” on page 755
- “tppc\_prepare\_to\_receive—Change to Receive State” on page 768.

## tppc\_activate\_on\_receipt—Activate a Program after Information Received

This function is an extension to the LU 6.2 architecture. This function allows the issuing TPF ECB to exit and activate a different ECB at the program specified after information has been received. The information received may be data, conversation status, or a confirmation request.

### Format

```
#include <tppc.h>
void tppc_activate_on_receipt(unsigned int *resid,
                             struct tppc_return_codes *rcode,
                             enum t_aor_dl level,
                             unsigned char *field,
                             unsigned char *pgm);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

#### level

This specifies a data level that contains a working storage block to be passed to the activated program. After the information is received, the contents of the block on the data level specified is passed from the ECB issuing this verb to a 4K block on the same data level of the ECB created. This argument must belong to the enumeration type t\_aor\_dl, defined in tppc.h. Use a value in the range of **AOR\_D1** to **AOR\_DA**. If no block will be passed, use the value **AOR\_NO\_DL**.

#### field

This is a pointer to an 8-byte field that is passed to the program specified by **pgm** when it is activated. The data in this field is copied to EBX000 of the new ECB that is created after the information is received. If the **level** parameter is used instead, this parameter must have a value of **NULL**.

#### pgm

This is a pointer to a 4-byte field that contains the name of the TPF real-time program to be activated after information is received. This TPF real-time segment must be defined in the TPF transaction program name table (TPNT) as an ACTIVATE\_ON\_RECEIPT target program. This is done with the ITPNT macro by specifying TYPE=AOR and by specifying the segment name as both the TPN name and the PGM name. See *TPF General Macros* for more information about the ITPNT macro.

### Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the tppc\_activate\_on\_receipt function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	

## tppc\_activate\_on\_receipt

Symbolic Name	Primary Code	Secondary Code
LU62RC_AOR_INCOMPLETE	....	00000009
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_PK_BAD_PROG	....	000000F2
LU62RC_PK_BAD_PARM	....	000000F3
LU62RC_STATE_CHECK	0002	
LU62RC_SKRECEV_BADSTATE	....	000000B1
LU62RC_TPF_ABEND	FFFF	

The following table contains a list of the primary and secondary return codes that can be passed to the program specified by **pgm** when it is activated. The return codes are based on the processing of the RECEIVE function.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_ALLOC_ERROR	0003	
LU62RC_TP_NOT_AVAIL_RETRY	....	084B6031
LU62RC_TP_NOT_AVAIL_NO_RETRY	....	084C0000
LU62RC_TPN_NOT_RECOGNIZED	....	10086021
LU62RC_PIP_NOT_SPECIFIED_CORRECTLY	....	10086032
LU62RC_CONV_TYPE_MISMATCH	....	10086034
LU62RC_SYNLVL_NOTSUPPORT	....	10086041
LU62RC_DLLOC_ABEND_PGM	0006	
LU62RC_DLLOC_ABEND_SVC	0007	
LU62RC_DLLOC_ABEND_TMR	0008	
LU62RC_DLLOC_NORMAL	0009	
LU62RC_PGMERR_NOTRUNC	000C	
LU62RC_PGMERR_TRUNC	000D	
LU62RC_PGMERR_PURGING	000E	
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NO_RETRY	0010	
LU62RC_SVCERR_NOTRUNC	0011	
LU62RC_SVCERR_TRUNC	0012	
LU62RC_SVCERR_PURGING	0013	
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The conversation must be in send or receive state.
- The value supplied in **resid** must be the resource ID returned by the `tppc_allocate` function or one that was assigned by an incoming ATTACH.

- You can call this function instead of the `tppc_receive` function. If there is no information available to satisfy a RECEIVE when the `tppc_activate_on_receipt` function is called, an ECB is not suspended while waiting for the information to arrive. When information arrives, TPF/APPC support creates a new ECB, performs a RECEIVE, and activates the program specified by **pgm**. The information that was received is passed to the activated program.
- When the program specified by **pgm** is activated, this indicates that a RECEIVE was done. The return codes are passed to the program in contiguous memory starting at EBX008. Check these codes to determine the success of the RECEIVE. If the return code is LU62RC\_OK, check the WHAT\_RECEIVED field (passed at EBX014) to see what was received. If WHAT\_RECEIVED is DATA (COMPLETE or INCOMPLETE), the data is passed to the program on D0.

**Note:** The structure of the `tppc_return_codes` requires 2 alignment bytes between the primary and secondary return codes. This structure cannot be used to overlay the area starting at EBX008.

- If there is information available when the `tppc_activate_on_receipt` function is called, TPF/APPC activates the program specified by **pgm** right away with a new ECB. The information is not returned to the program that calls this function.
- If the return code is LU62RC\_OK, the ECB is no longer connected to the transaction program or the conversation. The ECB cannot issue any more TPF/APPC verbs.
- If the return code is any value other than LU62RC\_OK, the ECB is still connected to the transaction program and the conversation. However, because all other return codes indicate a serious error, this also indicates that a DEALLOCATE ABEND occurred for the conversation. The ECB cannot issue any more TPF/APPC verbs for this conversation but can issue TPF/APPC verbs for other conversations associated with the transaction program instance.

**Note:** The transaction program is identified with the transaction program identifier (TCB ID), and the conversation is identified with the conversation control block identifier (CCB ID). Both identifiers are defined in the ECB as EBTCBID and EBCCBID respectively.

- You must specify either the **level** parameter or the **field** parameter, but not both.
- Any working storage block passed on a data level to the activated program is released by TPF/APPC.
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

```
#include <tppc.h>

        unsigned int      resource_id;
        struct tppc_return_codes return_code;
        unsigned char      field[8];
        unsigned char      pgm[4];
        :
tppc_activate_on_receipt(&resource_id,&return_code,          \
                        AOR_NO_DL,field,pgm);
```

**tppc\_activate\_on\_receipt**

⋮

/\* Normal processing path

\*/

## Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_allocate—Allocate a Conversation” on page 745
- “tppc\_receive—Receive Information” on page 771.

## tppc\_allocate—Allocate a Conversation

This function allocates a conversation between a TPF transaction program and a transaction program in a remote LU.

The function assigns a resource ID (RESID) to the conversation. You must specify this resource ID on all subsequent functions for this conversation.

### Format

```
#include <tppc.h>
void tppc_allocate(unsigned int *resid,
                   struct tppc_return_codes *rcode,
                   struct tppc_name *luname,
                   struct tppc_proname *tpn,
                   unsigned char *mode,
                   enum t_allocate_sync sync,
                   enum t_allocate_rcontrol rcontrol,
                   enum t_allocate_type type,
                   enum t_allocate_pip pip,
                   enum t_allocate_security security);
```

#### resid

This is a pointer to a 4-byte field where the resource ID is returned. You must specify this resource ID on all subsequent functions for this conversation.

#### rcode

This is a pointer to the structure `tppc_return_codes`, defined in `tppc.h`, where the return code is to be placed.

#### luname

This is a pointer to the structure `tppc_name`, defined in `tppc.h`. This structure contains the network name of the remote LU or local secondary LU (SLU) thread with which this local transaction program wants to start a conversation. The first 8 bytes contain the left-justified network name, which is padded with blanks, or all blanks if the name is unqualified. The second 8 bytes contain the left-justified LU name, which is padded with blanks.

#### tpn

This is a pointer to the structure `tppc_proname`, defined in `tppc.h`, which contains the following:

- 1-byte length of the remote transaction program name
- A remote transaction program name, which can be from 1–64 characters long.

#### mode

This is a pointer to an 8-byte field that contains the mode name, which designates the properties of the session to be allocated.

#### sync

This specifies the synchronization level allowed on this conversation. This argument must belong to the enumeration type `t_allocate_sync`, defined in `tppc.h`. The allowed values are:

##### ALLOCATE\_SYNC\_NONE

This value specifies that the programs cannot perform confirmation processing on this conversation.

##### ALLOCATE\_SYNC\_CONFIRM

This value specifies that the programs can perform confirmation processing on this conversation.

## **tppc\_allocate**

**Note:** The TPF system does not support the SYNCPT option of the LU 6.2 architecture.

### **rcontrol**

This argument specifies when control is returned to the caller. This argument must belong to the enumeration type `t_allocate_rcontrol`, defined in `tppc.h`. The allowed values are:

#### **ALLOCATE\_RCONTROL\_WSA**

This value specifies that control is returned when a session is allocated for this conversation.

#### **ALLOCATE\_RCONTROL\_IMM**

This value specifies to allocate a session for the conversation if a session is immediately available. A session is immediately available when it is active, it is not allocated to another conversation, and the local LU is the contention winner for the session.

**Note:** The TPF system does not support the other options defined for this parameter by the LU 6.2 architecture.

### **type**

This argument must belong to the enumeration type `t_allocate_type`, defined in `tppc.h`. The allowed values are:

#### **ALLOCATE\_TYPE\_BASIC**

This value provides support for the BASIC\_CONVERSATION option defined by the LU 6.2 architecture.

#### **ALLOCATE\_TYPE\_MAPPED**

This value provides support for the MAPPED\_CONVERSATION option defined by the LU 6.2 architecture. This is provided for use with your own mapped support. See “TPF/APPC Mapped Conversation Functions” on page 793 for information about TPF’s mapped conversation support.

#### **ALLOCATE\_TYPE\_SHAREDB**

This value provides support for the BASIC\_CONVERSATION option defined by the LU 6.2 architecture, and specifies this is a shared LU 6.2 conversation.

#### **ALLOCATE\_TYPE\_SHAREDM**

This value provides support for the MAPPED\_CONVERSATION option defined by the LU 6.2 architecture, and specifies this is a shared LU 6.2 conversation.

### **pip**

This argument must belong to the enumeration type `t_allocate_pip`, defined in `tppc.h`. Use the value **ALLOCATE\_PIP\_NO**. PIP (program initialization parameters) data cannot be supplied by the TPF transaction program. If PIP data is indicated in an ATTACH header received by the TPF system, the ATTACH conversation request is rejected.

### **security**

This specifies the security level allowed on this conversation. This argument must belong to the enumeration type `t_allocate_security`, defined in `tppc.h`. Use the value **ALLOCATE\_SECURITY\_NO**.

**Note:** The TPF system does not support the SAME and PGM options supported by the LU 6.2 architecture.



## Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the `tppc_allocate` function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_OPTION	....	00C62074
LU62RC_ALLOC_ERROR	0003	
LU62RC_ALLOCERR_NORETRY	....	00000004
LU62RC_ALLOCERR_RETRY	....	00000005
LU62RC_INVALID_MODE_NAME	....	000000F3
LU62RC_ALLOC_UNSUCESSFUL	0015	
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The value returned in **resid** must be used by all other TPF/APPC functions called for this conversation.
- If the value of the **type** parameter is **ALLOCATE\_TYPE\_SHAREDDB** or **ALLOCATE\_TYPE\_SHAREDM**, then this is a shared LU 6.2 conversation. Shared LU 6.2 conversations are one-way pipes used to send data from the TPF system to a remote LU. Verbs can be issued from any ECB for a shared conversation. For conversations that are not shared, verbs can be issued only from the ECB that started the conversation. Because a shared conversation is a one-way pipe used to send data, only certain LU 6.2 verbs are allowed:
  - `tppc_send_data`
  - `tppc_flush`
  - `tppc_get_attributes`
  - `tppc_deallocate` (except when **DEALLOCATE\_TYPE\_CONFIRM** is specified on the **type** parameter).

See *TPF ACF/SNA Data Communications Reference* for more information about shared LU 6.2 conversations.

- The value returned in **resid** is stored in field EBCCBID in the ECB. This value changes with every **ALLOCATE** request.
- If you are allocating a secondary LU (SLU) thread session, you must specify a single session mode name. Parallel sessions are not supported for SLU threads.
- The **ATTACH** is buffered until the buffer is full or a verb that implies the `tppc_flush` function is issued.
- If the session assigned to this conversation is a contention winner, the session is assigned without seeking permission from the remote partner. If the value for **rcontrol** is **ALLOCATE\_RCONTROL\_WSA** and if the session assigned to this conversation is a contention loser, permission is sought from the remote partner before the allocation is permitted.
- A session is activated for this conversation if all of the following conditions are true:
  - The **rcontrol** parameter is **ALLOCATE\_RCONTROL\_WSA**
  - A session is not already available for the conversation

## tppc\_allocate

- The session limits have **not** been reached for this **luname** and **mode**.

**Note:** PU 2.1 SLU thread sessions cannot be activated from the TPF side. In this case, if you specify a local SLU thread, that SLU must already be in session with the remote LU.

This verb uses the TPF system's EVENT and POST facility to suspend the ECB until a session is established.

If a session is not established in a certain amount of time, the program sends a failure return code to the transaction program. The amount of time that the system waits is determined by the value you specify for the TPALLOC parameter on the SNAKEY macro. See *TPF ACF/SNA Network Generation* for information about the SNAKEY macro.

ALLOCATE must know what 2 LUs are involved to activate the session. One LU is specified with **luname**; the other LU is determined as follows:

- If you specify a remote LU that is in session with SLU threads, ALLOCATE selects the thread to be used for the session.
- If you specify a local LU that is a SLU thread, a remote LU must have been previously initialized for that SLU thread (with a CNOS INITIALIZE request).
- If you specify a parallel sessions mode name, the LU specified on the CNOS INITIALIZE request is used.
- The default TPF/APPC LU is used if the following conditions are true:
  - You specify single sessions
  - You are not using SLU threads
  - A CNOS INITIALIZE was not done.
- The LU specified on the CNOS INITIALIZE is used if the following conditions are true:
  - You specify single sessions
  - A CNOS INITIALIZE was done.
- The ALLOCATE request is queued until a session becomes available for use by this conversation if all of the following conditions are true:
  - The **rcontrol** parameter is **ALLOCATE\_RCONTROL\_WSA**
  - A session is not already available for the conversation
  - The session limits have been reached for this **luname** and **mode**.
- If the conversation is allocated to a contention loser session, this verb uses the TPF system's EVENT and POST facility to suspend the ECB until the program receives a BID response.

If the program does not receive a BID response from the remote LU in a certain amount of time, an UNBIND is scheduled for this session, and the program sends a failure return code to the transaction program. The amount of time that the system waits is determined by the value you specify for the TPALLOC parameter on the SNAKEY macro. See *TPF ACF/SNA Network Generation* for information about the SNAKEY macro.

- If the ECB that issues tppc\_allocate does not already have a TCBID stored in field EBTCBID in the ECB, the ALLOCATE request represents a new TPF transaction program instance, and a new TCBID is stored in EBTCBID. If EBTCBID already has a value stored, the ALLOCATE request represents another conversation for the same transaction program instance, and EBTCBID is not changed.
- The mode name SNASVCMG cannot be used by a user TPF transaction program.

- When the ALLOCATE is completed, the conversation on the local transaction program side is in send state, and the conversation on the remote transaction program side in receive state.
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example establishes a conversation between a program in the local TPF LU and a remote LU on an LU-LU session that is already established.

```
#include <tppc.h>

    unsigned int          resource_id;
    struct tppc_return_codes return_code;
    struct tppc_name       their_lu_name;
    struct tppc_proname    their_tp_name;
    unsigned char          modename[8] = "TPFLU62";
    :
tppc_allocate(&resource_id,&return_code,&their_lu_name, \
    &their_tp_name,modename,ALLOCATE_SYNC_NONE,ALLOCATE_RCONTROL_WSA, \
    ALLOCATE_TYPE_BASIC,ALLOCATE_PIP_NO,ALLOCATE_SECURITY_NO);
    :
```

## Related Information

“Return Codes for Basic Conversation Functions” on page 731.

## tppc\_confirm–Send a Confirmation Request

This function sends a confirmation request to the remote transaction program and waits for the confirmation reply. This allows the 2 programs to synchronize their processing.

### Format

```
#include <tppc.h>
void tppc_confirm(unsigned int *resid,
                  struct tppc_return_codes *rcode,
                  enum t_rtsrcvd *rtsrcvd);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

#### rtsrcvd

This is a pointer to a 1-byte field that contains the enumeration type t\_rtsrcvd, where the REQUEST\_TO\_SEND\_RECEIVED indication is placed. The REQUEST\_TO\_SEND\_RECEIVED indication is one of the following:

##### RTSRCVD\_YES

This value indicates that a REQUEST\_TO\_SEND indication was received from the remote transaction program. The remote program called tppc\_request\_to\_send requesting the local TPF transaction program to enter receive state and placing the remote transaction program in send state.

##### RTSRCVD\_NO

This value indicates that a REQUEST\_TO\_SEND notification was not received.

### Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the tppc\_confirm function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_STATE_CHECK	0002	
LU62RC_SKCNFRM_BADSTATE	....	00000032
LU62RC_SKCNFRM_INVALID	....	00000033
LU62RC_ALLOC_ERROR	0003	
LU62RC_SECURITY_NOT_VALID	....	080F6051
LU62RC_TP_NOT_AVAIL_RETRY	....	084B6031

Symbolic Name	Primary Code	Secondary Code
LU62RC_TP_NOT_AVAIL_NO_RETRY	....	084C0000
LU62RC_TPN_NOT_RECOGNIZED	....	10086021
LU62RC_PIP_NOT_SPECIFIED_CORRECTLY	....	10086032
LU62RC_CONV_TYPE_MISMATCH	....	10086034
LU62RC_SYNLVL_NOTSUPPORT	....	10086041
LU62RC_DLLOC_ABEND_PGM	0006	
LU62RC_DLLOC_ABEND_SVC	0007	
LU62RC_DLLOC_ABEND_TMR	0008	
LU62RC_PGMERR_PURGING	000E	
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NORETRY	0010	
LU62RC_SVCERR_PURGING	0013	
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The conversation must be in send state.
- The value supplied in **resid** must be the resource ID returned by the tppc\_allocate function or one that was assigned by an incoming ATTACH.
- This verb uses TPF's EVENT and POST facility to suspend the ECB until the program receives a confirmation reply. Since the ECB is suspended, you should release all unnecessary resources before issuing this verb. Failure to do so can cause serious system performance degradation.

**Note:** You can avoid the problem of suspended ECBs by using the tppc\_activate\_on\_confirmation function instead of tppc\_confirm. See “tppc\_activate\_on\_confirmation—Activate a Program after Confirmation Received” on page 737 for more information.

- If the program does not receive a confirmation reply in a certain amount of time, TPF/APPC support issues a tppc\_deallocate with **DEALLOCATE\_TYPE\_ABENDP** to terminate the conversation. The amount of time that the system waits is determined by the value you specify for the TPrecv parameter on the SNAKEY macro. See *TPF ACF/SNA Network Generation* for information about the SNAKEY macro.
- When the **rtsrcvd** parameter is **RTSRCVD\_YES**, the remote program is requesting the local TPF transaction program to enter receive state and thereby place the remote program in send state. The local TPF transaction program enters receive state by issuing tppc\_receive or tppc\_prepare\_to\_receive. The remote partner program enters the corresponding send state when it calls tppc\_receive and receives the SEND indicator on the **whatrcv** parameter.
- If **rcode** is LU62RC\_PGMERR\_PURGING or LU62RC\_SVCERR\_PURGING, the conversation enters receive state.
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## tppc\_confirm

### Examples

This function sends a confirmation request to the remote transaction program and waits for the confirmation reply.

```
#include <tppc.h>

    unsigned int      resource_id;
    struct tppc_return_codes return_code;
    enum t_rtsrcvd     request_received;
    :
    :
/* set up resource_id with the value returned from the allocate verb */
    :
    :
tppc_confirm(&resource_id,&return_code,&request_received);
    :                               /* normal processing path          */
    :
```

### Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_confirmed—Send a Confirmation Reply” on page 753
- “tppc\_allocate—Allocate a Conversation” on page 745.

## tppc\_confirmed–Send a Confirmation Reply

This function sends a confirmation reply to the remote transaction program. This allows the 2 programs to synchronize their processing. The local TPF transaction program can call this function when it receives a confirmation request from the remote transaction program. See the **WHAT\_RECEIVED (whatrcv)** parameter of the `tppc_receive` function (“tppc\_receive–Receive Information” on page 771).

### Format

```
#include <tppc.h>
void tppc_confirmed(unsigned int *resid,
                    struct tppc_return_codes *rcode);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial `ALLOCATE` for this conversation or one that was assigned by an incoming `ATTACH`.

#### rcode

This is a pointer to the structure `tppc_return_codes`, defined in `tppc.h`, where the return code is to be placed.

### Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the `tppc_confirmed` function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_STATE_CHECK	0002	
LU62RC_SKCNFMD_BADSTATE	....	00000041
LU62RC_TPF_ABEND	FFFF	

### Programming Considerations

- The conversation must be in one of the following states:  
received-confirm  
received-confirm-send  
received-confirm-deallocate.
- The value supplied in **resid** must be the resource ID returned by the `tppc_allocate` function or one that was assigned by an incoming `ATTACH`.
- A transaction program can call this function only as a reply to a confirmation request; the function cannot be called at any other time. Transaction programs can use this function for various application-level functions. For example, the remote program can send data followed by a confirmation request. When the local program receives the confirmation request, it can call this function as an indication that it received and processed the data without error.

## tppc\_confirmed

- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example sends a confirmation reply to the remote transaction program.

```
#include <tppc.h>

    unsigned int      resource_id;
    struct tppc_return_codes return_code;
    :
    :
/* set up resource_id with the value returned from the allocate verb */
    :
    tppc_confirmed(&resource_id,&return_code);
    :                               /* normal processing path          */
    :
```

## Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_confirm—Send a Confirmation Request” on page 750
- “tppc\_allocate—Allocate a Conversation” on page 745
- “tppc\_receive—Receive Information” on page 771.



## tppc\_deallocate—Deallocate a Conversation

This function deallocates the specified conversation from the transaction program. An implied FLUSH is executed, and the resource ID becomes unassigned when the deallocation is complete.

### Format

```
#include <tppc.h>
void tppc_deallocate(unsigned int *resid,
                    struct tppc_return_codes *rcode,
                    enum t_deallocate_type type,
                    enum t_deallocate_logdata logdata);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

#### type

This argument must belong to the enumeration type t\_deallocate\_type, defined in tppc.h. Use one of the following values:

##### DEALLOCATE\_TYPE\_SYNC

This value specifies that either the tppc\_flush or the tppc\_confirm function should be performed before the conversation is deallocated, depending on the **sync** level specified at ALLOCATE time.

##### DEALLOCATE\_TYPE\_FLUSH

This specifies that the function of the FLUSH verb should be executed and then the conversation should be deallocated.

##### DEALLOCATE\_TYPE\_CONFIRM

This specifies that the function of the CONFIRM verb should be executed and then the conversation should be deallocated.

##### DEALLOCATE\_TYPE\_LOCAL

This specifies that the conversation should be deallocated locally. This type of deallocation can be specified **only** if the conversation is already in end-conversation state.

##### DEALLOCATE\_TYPE\_ABENDP

##### DEALLOCATE\_TYPE\_ABENDS

##### DEALLOCATE\_TYPE\_ABENDT

The 3 abend parameters specify that the conversation should be unconditionally deallocated. Logical record truncation can occur when the conversation is in send state, and data purging can occur when the conversation is in receive state. The only difference among the 3 abend codes is in the sense codes used to notify the remote transaction program. **ABENDP** is intended to be used by the application transaction program to indicate that it is requesting the deallocation. **ABENDS** and **ABENDT** are intended to be used by the TPF/APPC support routines when they request the deallocation.

## tppc\_deallocate

### logdata

This specifies whether error information should be logged. This argument must belong to the enumeration type `t_deallocate_logdata`, defined in `tppc.h`. Use the value **DEALLOCATE\_LOGDATA\_NO**.

**Note:** The TPF system does not support the value **YES** that is defined by the LU 6.2 architecture.

## Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the `tppc_deallocate` function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_PKDLLOC_BADTYPE	....	00000051
LU62RC_PK_BAD_OPTION	....	00C62074
LU62RC_STATE_CHECK	0002	
LU62RC_SKDLLOC_FLUSH	....	00000052
LU62RC_SKDLLOC_CONFIRM	....	00000053
LU62RC_SKDLLOC_ABEND	....	00000056
LU62RC_SKDLLOC_LOCAL	....	00000057
LU62RC_ALLOC_ERROR	0003	
LU62RC_SECURITY_NOT_VALID	....	080F6051
LU62RC_TP_NOT_AVAIL_RETRY	....	084B6031
LU62RC_TP_NOT_AVAIL_NO_RETRY	....	084C0000
LU62RC_TPN_NOT_RECOGNIZED	....	10086021
LU62RC_PIP_NOT_SPECIFIED_CORRECTLY	....	10086032
LU62RC_CONV_TYPE_MISMATCH	....	10086034
LU62RC_SYNLVL_NOTSUPPORT	....	10086041
LU62RC_DLLOC_ABEND_PGM	0006	
LU62RC_DLLOC_ABEND_SVC	0007	
LU62RC_DLLOC_ABEND_TMR	0008	
LU62RC_PGMERR_PURGING	000E	
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NORETRY	0010	
LU62RC_SVCERR_PURGING	0013	
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The conversation must be in the following state, depending on the type of deallocation requested:

Type	State Required
<b>FLUSH</b>	<u>send</u>
<b>CONFIRM</b>	<u>send</u>
<b>LOCAL</b>	<u>end-conversation</u>
<b>ABEND</b>	<u>send</u> , <u>receive</u> , <u>received-confirm</u> , <u>received-confirm-send</u> , or <u>received-confirm-deallocate</u> .

- The value supplied in **resid** must be the resource ID returned by the `tppc_allocate` function or one that was assigned by an incoming ATTACH.
- When the DEALLOCATE is complete, the conversation identified by the resource ID is completed, and no further functions can be called for that conversation.
- If the transaction program exits without deallocating a conversation, the TPF system does not deallocate that conversation. The program should call this function for all conversations before exiting.
- If the **type=DEALLOCATE\_TYPE\_CONFIRM** option is specified, TPF's EVENT and POST facility is used to suspend the ECB until the program receives a confirmation reply. Since the ECB is suspended, all unnecessary resources should be released before the issuance of this verb. Failure to do so can cause serious system degradation.

**Note:** You can avoid the problem of suspended ECBs by using the `tppc_activate_on_confirmation` function instead of `tppc_deallocate`. See “`tppc_activate_on_confirmation`—Activate a Program after Confirmation Received” on page 737 for more information.

- If the **type=DEALLOCATE\_TYPE\_CONFIRM** option is specified, the normal response from the remote transaction program is CONFIRMED. However, if the remote program issues a DEALLOCATE TYPE=ABEND, the local transaction program goes into end-conversation state. If the remote transaction program issues a SEND\_ERROR, the local program goes into receive state, and the conversation continues.
- If the **type=DEALLOCATE\_TYPE\_CONFIRM** option is issued and the program does not receive a confirmation reply within a certain amount of time, TPF/APPC support issues an unbind to terminate the conversation. The amount of time that the system waits is determined by the value you specify for the TPRECV parameter on the SNAKEY macro. See *TPF ACF/SNA Network Generation* for information about the SNAKEY macro.
- The remote transaction program receives the deallocation notification by either a return code or the WHAT\_RECEIVED indication. (See the **whatrcv** parameter of the `tppc_receive` function; “`tppc_receive`—Receive Information” on page 771.)
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example deallocates a specified conversation.

```
#include <tppc.h>
```

```
    unsigned int      resource_id;
    struct tppc_return_codes return_code;
```

## **tppc\_deallocate**

```

:
/* set up resource_id with the value returned from the allocate verb */
:
tppc_deallocate(&resource_id,&return_code,DEALLOCATE_TYPE_SYNC, \
                DEALLOCATE_LOGDATA_NO);
/* normal processing path */
:
```

## **Related Information**

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_allocate—Allocate a Conversation” on page 745
- “tppc\_post\_on\_receipt—Set Posting Active for a Conversation” on page 765.

## tppc\_flush–Flush Data from Local LU Buffer

This function allows the application to transmit any data from the data buffer of the local LU, including an ATTACH from the tppc\_allocate function.

### Format

```
#include <tppc.h>
void tppc_flush(unsigned int *resid,
                struct tppc_return_codes *rcode);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

### Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the tppc\_flush function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_STATE_CHECK	0002	
LU62RC_SKFLUSH_BADSTATE	....	00000061
LU62RC_TPF_ABEND	FFFF	

### Programming Considerations

- The conversation must be in send state.
- The value supplied in **resid** must be the resource ID returned by the tppc\_allocate function or one that was assigned by an incoming ATTACH.
- This function is useful for optimization of processing between the local and remote transaction programs. The TPF/APPC support code buffers the information from consecutive tppc\_send\_data functions until it has a sufficient amount of information for transmission. At that time, it transmits the buffered data. However, the local TPF transaction program can issue tppc\_flush in order to cause the TPF/APPC support code to transmit any buffered data. In this way, the local program can minimize the delay in the remote transaction program's processing. The TPF/APPC support code actually flushes the buffered data only when it has some data buffered; if there is no data buffered, nothing is transmitted to the remote LU. The buffer size is determined by the maximum request unit (RU) size of the session, which is negotiated at BIND time.
- See "Programming Considerations for Basic Conversation Functions" on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## tppc\_flush

### Examples

The following example invokes the `tppc_flush` function.

```
#include <tppc.h>

    unsigned int      resource_id;
    struct tppc_return_codes return_code;
    :
/* set up resource_id with the value returned from the allocate verb */
    :
    tppc_flush(&resource_id,&return_code);
                                /* normal processing path      */
    :
```

### Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “`tppc_allocate`—Allocate a Conversation” on page 745.

## tppc\_get\_attributes—Get Information about a Conversation

This function returns information pertaining to a conversation.

### Format

```
#include <tppc.h>
void tppc_get_attributes(unsigned int *resid,
                        struct tppc_return_codes *rcode,
                        struct tppc_name *ownname,
                        struct tppc_name *pluname,
                        unsigned char *mode,
                        enum t_get_attributes_sync *sync);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

#### ownname

This is a pointer to the structure tppc\_name, defined in tppc.h, where the 16-byte network name of the local TPF LU is returned. The first 8 bytes contain the left-justified network name, which is padded with blanks, or all blanks if the name is unqualified. The second 8 bytes contain the left-justified local LU name, which is padded with blanks.

#### pluname

This is a pointer to the structure tppc\_name, defined in tppc.h, where the 16-byte network name of the partner (remote) LU is returned. The first 8 bytes contain the left-justified network name, which is padded with blanks, or all blanks if the name is unqualified. The second 8 bytes contain the left-justified partner LU name, which is padded with blanks. This returned LU name is the name of the LU where the remote transaction program is located.

#### mode

This is a pointer to an 8-byte field where the mode name of the conversation is returned.

#### sync

This is a pointer to a 1-byte field that contains the enumeration type t\_get\_attributes\_sync, where the synchronization level of this conversation is returned. The returned values are:

##### GET\_ATTRIBUTES\_SYNC\_NONE

This value indicates that no synchronization is allowed on this conversation.

##### GET\_ATTRIBUTES\_SYNC\_CONFIRM

This value indicates that CONFIRM synchronization is allowed on this conversation.

**Note:** The TPF system does not support the SYNCPT level defined by the LU 6.2 architecture.

### Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the tppc\_get\_attributes function. A complete list of the return codes and their definitions can be found in Table 37 on page 731

## tppc\_get\_attributes

page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_PK_BAD_OPTION	....	00C62074
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The conversation can be in any state.
- The value supplied in **resid** must be the resource ID returned by the `tppc_allocate` function or one that was assigned by an incoming ATTACH.
- If any of the **ownname**, **pluname**, **mode**, or **sync** parameters are null pointers (zero), that parameter's information is not returned. However, if all of the parameters are null, an error is issued.
- The TPF system does not support the other parameters defined by the LU 6.2 architecture for the GET\_ATTRIBUTES verb.
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example retrieves the attributes of a conversation.

```
#include <tppc.h>

    unsigned int      resource_id;
    struct tppc_return_codes return_code;
    struct tppc_name   own_name;
    struct tppc_name   plu_name;
    unsigned char      mode_name[8];
    enum t_get_attributes_sync sync_level;
    :
    :
/* set up resource_id with the value returned from the allocate verb */
    :
    :
    tppc_get_attributes(&resource_id,&return_code,&own_name,&plu_name, \
        mode_name,&sync_level);
    :                               /* normal processing path          */
    :
```

## Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_allocate—Allocate a Conversation” on page 745.



## tppc\_get\_type—Get Conversation Type

This function returns the conversation type, which can be either basic or mapped.

### Format

```
#include <tppc.h>
void tppc_get_type(unsigned int *resid,
                  struct tppc_return_codes *rcode,
                  enum t_get_type_type *type);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in **tppc.h**, where the return code is to be placed.

#### type

This is a pointer to a 1-byte field that contains the enumeration type t\_get\_type\_type, where the conversation type is to be returned. The returned values are:

#### GET\_TYPE\_BASIC\_CONVERSATION

This value indicates that the resource is a basic conversation.

#### GET\_TYPE\_MAPPED\_CONVERSATION

This value indicates that the resource is a mapped conversation.

### Return Codes

The return codes that can be returned to the program that called the tppc\_get\_type function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002

### Programming Considerations

- The conversation can be in any state.
- The value supplied in **resid** must be the resource ID returned by the tppc\_allocate function or one that was assigned by an incoming ATTACH.
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

### Examples

The following example retrieves the conversation type.

```
#include <tppc.h>

unsigned int resource_id;
struct tppc_return_codes return_code;
```

## **tppc\_get\_type**

```
        enum      t_get_type_type      type;
        :
        :
        /* set up resource_id with the value returned from the allocate verb */
        :
        :
        tppc_get_type(&resource_id,&return_code,&type);
        :                               /* normal processing path          */
        :
```

## **Related Information**

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_allocate—Allocate a Conversation” on page 745.

## tppc\_post\_on\_receipt–Set Posting Active for a Conversation

This function causes the conversation to be posted when information is available for the transaction program to receive. The information may be data or conversation information. The `tppc_wait` function should be called after this function in order to wait for posting to occur. Alternatively, `tppc_test` can be called after this function in order to determine when posting has occurred.

### Format

```
#include <tppc.h>
void tppc_post_on_receipt(unsigned int *resid,
                        struct tppc_return_codes *rcode,
                        enum t_post_on_receipt_fill fill,
                        unsigned short int *length);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial `ALLOCATE` for this conversation or one that was assigned by an incoming `ATTACH`.

#### rcode

This is a pointer to the structure `tppc_return_codes`, defined in `tppc.h`, where the return code is to be placed.

**fill** This argument must belong to the enumeration type `t_post_on_receipt_fill`, defined in `tppc.h`. Use the value **POST\_ON\_RECEIPT\_FILL\_LL**, which specifies that posting occurs when a complete or truncated logical record is received, or when a part of a logical record is received that is at least equal in length to that specified on the **length** parameter.

**Note:** The TPF system does not support the `BUFFER` option defined by the LU 6.2 architecture.

#### length

This is a pointer to a 2-byte field that specifies the minimum amount of data the LU must receive before the conversation can be posted. Use **length** along with the **fill** argument to determine when to post the conversation for the receipt of data. If the value is a `NULL` pointer, the conversation is posted when a complete logical record is received or other, nondata, information is received (such as `CONFIRM` or `PREPARE_TO_RECEIVE`).

### Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the `tppc_post_on_receipt` function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_PK_BAD_OPTION	....	00C62074
LU62RC_INVALID_LENGTH	....	00000006
LU62RC_STATE_CHECK	0002	

## tppc\_post\_on\_receipt

Symbolic Name	Primary Code	Secondary Code
LU62RC_SKPOSTR_BADSTATE	....	00000092
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The conversation must be in receive state.
- The value supplied in **resid** must be the resource ID returned by the `tppc_allocate` function or one that was assigned by an incoming ATTACH.
- This verb is intended for use in conjunction with the `tppc_wait` or `tppc_test` function. Use this function followed by a `tppc_wait` function to allow a transaction program to perform synchronous receiving from multiple conversations. For each conversation, the program issues this verb and then issues the `tppc_wait` function. This causes the conversations to wait until information is available to be received.

Use this function followed by a `tppc_test` function to allow a transaction program to continue its processing and test the conversation to determine when information is available to be received.

- Posting becomes active for a conversation when `tppc_post_on_receipt` is called for a conversation. Posting is reset when one of the following functions is called for a conversation **after** the conversation is posted:

```
tppc_deallocate
tppc_receive
tppc_send_error
tppc_test
tppc_wait.
```

Posting is canceled when one of the following functions is called for a conversation **before** the conversation is posted:

```
tppc_deallocate
tppc_send_error.
```

- Posting occurs when a complete logical record is available for the conversation, when the amount of data specified on the **length** parameter is available, or when information other than data is received, such as a confirmation request or an error indication. See the WHAT\_RECEIVED (**whatrcv**) parameter of the `tppc_receive` function for more information. (See “`tppc_receive`—Receive Information” on page 771.)
- If the **length** parameter is used, the maximum length that can be entered is 32,767.
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example causes the conversation to be posted when information is available for the transaction program to receive.

```
#include <tppc.h>

    unsigned int          resource_id;
    struct tppc_return_codes return_code;
    unsigned short int     length;
    :
/* set up resource_id with the value returned from the allocate verb */
```

```

:
tppc_post_on_receipt(&resource_id,&return_code,POST_ON_RECEIPT_FILL_LL, \
&length);
/* normal processing path */
:
```

## Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_allocate—Allocate a Conversation” on page 745
- “tppc\_wait—Wait for Posting” on page 789
- “tppc\_test—Test Conversation” on page 786.

## tppc\_prepare\_to\_receive—Change to Receive State

This function changes the conversation from send to receive state in preparation to receive data. The change to receive state is completed as part of this function. The execution of this function includes the function of the FLUSH or CONFIRM verb.

### Format

```
#include <tppc.h>
void tppc_prepare_to_receive(unsigned int *resid,
                             struct tppc_return_codes *rcode,
                             enum t_prepare_to_receive_type type,
                             enum t_prepare_to_receive_locks locks);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

#### type

This specifies the type of PREPARE\_TO\_RECEIVE to be done on this conversation. This argument must belong to the enumeration type t\_prepare\_to\_receive\_type, defined in tppc.h. Use one of the following values:

##### PREP\_TO\_RECEIVE\_TYPE\_CONFIRM

This executes the function of the CONFIRM verb. When the PREPARE\_TO\_RECEIVE request completes successfully, the conversation enters receive state.

##### PREP\_TO\_RECEIVE\_TYPE\_SYNC

This executes the function of the FLUSH verb or the CONFIRM verb based on the synchronization level of the conversation. If the synchronization level is **NONE**, a FLUSH is performed. If the synchronization level is **CONFIRM**, a CONFIRM is performed.

#### locks

This argument must belong to the enumeration type t\_prepare\_to\_receive\_locks, defined in tppc.h. Use the value **PREP\_TO\_RECEIVE\_LOCKS\_SHORT**. This parameter is meaningful only when the **PREP\_TO\_RECEIVE\_TYPE\_CONFIRM** option is used or implied on the **type** parameter, and it causes control to be returned only after the confirmation reply was received. Although this parameter has no meaning with other options, it still must be coded.

**Note:** The TPF system does not support the LONG option defined by the LU 6.2 architecture.

### Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the tppc\_prepare\_to\_receive function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	

Symbolic Name	Primary Code	Secondary Code
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_PKPTRCV_INVTYPE	....	000000A1
LU62RC_PK_BAD_OPTION	....	00C62074
LU62RC_STATE_CHECK	0002	
LU62RC_SKPTRCV_BADSTATE	....	000000A3
LU62RC_ALLOC_ERROR	0003	
LU62RC_SECURITY_NOT_VALID	....	080F6051
LU62RC_TP_NOT_AVAIL_RETRY	....	084B6031
LU62RC_TP_NOT_AVAIL_NO_RETRY	....	084C0000
LU62RC_TPN_NOT_RECOGNIZED	....	10086021
LU62RC_PIP_NOT_SPECIFIED_CORRECTLY	....	10086032
LU62RC_CONV_TYPE_MISMATCH	....	10086034
LU62RC_SYNLVL_NOTSUPPORT	....	10086041
LU62RC_DLLOC_ABEND_PGM	0006	
LU62RC_DLLOC_ABEND_SVC	0007	
LU62RC_DLLOC_ABEND_TMR	0008	
LU62RC_PGMERR_PURGING	000E	
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NORETRY	0010	
LU62RC_SVCERR_PURGING	0013	
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The conversation must be in send state.
- The value supplied in **resid** must be the resource ID returned by the `tppc_allocate` function or one that was assigned by an incoming ATTACH.
- Upon successful completion of this function, the conversation is in receive state.
- If the **type=PREP\_TO\_RECEIVE\_TYPE\_CONFIRM** option is specified, TPF's EVENT and POST facility is used to suspend the ECB until the program receives the confirmation reply. Since the ECB is suspended, all unnecessary resources should be released before this verb is issued. Failure to do so can cause serious system degradation.

**Note:** You can avoid the problem of suspended ECBs by using the `tppc_activate_on_confirmation` function instead of `tppc_prepare_to_receive`. See "tppc\_activate\_on\_confirmation—Activate a Program after Confirmation Received" on page 737 for more information.

- If the **type=PREP\_TO\_RECEIVE\_TYPE\_CONFIRM** option is specified and the program does not receive a confirmation reply within a certain amount of time, TPF/APPC support issues a `tppc_deallocate` with **DEALLOCATE\_TYPE\_ABENDP** to terminate the conversation. The amount of time that the system waits is determined by the value you specify for the

## tppc\_prepare\_to\_receive

TPRECV parameter on the SNAKEY macro. See the *TPF ACF/SNA Network Generation* for information about the SNAKEY macro.

- The remote transaction program enters the corresponding send state when it receives the SEND indication in the WHAT\_RECEIVED (**whatrcv**) parameter. (See “tppc\_receive—Receive Information” on page 771) The remote transaction program can then send data to the local TPF transaction program.
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example changes the direction of the conversation.

```
#include <tppc.h>

    unsigned int      resource_id;
    struct tppc_return_codes return_code;
    :
    :
/* set up resource_id with the value returned from the allocate verb */
    :
    :
tppc_prepare_to_receive(&resource_id,&return_code,          \
    PREP_TO_RECEIVE_TYPE_SYNC,PREP_TO_RECEIVE_LOCKS_SHORT);
    :                               /* normal processing path          */
    :
```

## Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_allocate—Allocate a Conversation” on page 745.



## tppc\_receive—Receive Information

This function waits for information to arrive on the specified conversation and then receives the information. If information is already available, the information is received without waiting. The information received may be data, conversation status, or a confirmation request.

The transaction program can call this function when the conversation is in send state. In this case, the SEND indication is sent to the remote transaction program, and the local TPF side of the conversation is changed to receive state and waits for information to arrive. The remote program can send data to the local program after it receives the SEND indication.

## Format

```
#include <tppc.h>
void tppc_receive(unsigned int *resid,
                  struct tppc_return_codes *rcode,
                  enum t_receive_whatrcv *whatrcv,
                  enum t_rtsrcvd *rtsrcvd,
                  enum t_receive_fill fill,
                  enum t_receive_wait wait,
                  unsigned char *data,
                  unsigned short int *length);
```

### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

### whatrcv

This is a pointer to a 1-byte field that contains the enumeration type t\_receive\_whatrcv, where the WHAT\_RECEIVED indication is placed. The WHAT\_RECEIVED indication is one of the following:

#### RECEIVE\_WHATRCV\_DATACOMPLETE

This value indicates that a complete logical record or the last remaining portion of a logical record is received.

#### RECEIVE\_WHATRCV\_DATAINCOMPLETE

This value indicates that less than a complete logical record is received by the local TPF transaction program. This can be caused by the remote transaction program calling a function that causes data truncation, such as, tppc\_deallocate with one of the abend parameters.

#### RECEIVE\_WHATRCV\_LL\_TRUNCATED

This value indicates that the 2-byte length field of a logical record was truncated after the first byte. The TPF/APPC support code discards the truncated length field; it is not received by the program.

#### RECEIVE\_WHATRCV\_CONFIRM

This value indicates that the remote program called tppc\_confirm requesting the local TPF transaction program to respond by issuing tppc\_confirmed. The local TPF transaction program can instead call tppc\_send\_error.

#### RECEIVE\_WHATRCV\_CONFIRMSEND

This value indicates that the remote program called

## tppc\_receive

tppc\_prepare\_to\_receive with

**type=PREP\_TO\_RECEIVE\_TYPE\_CONFIRM**. The local TPF transaction program can respond by issuing `tppc_confirmed` or `tppc_send_error`.

### RECEIVE\_WHATRCV\_SEND

This value indicates that the remote program has entered receive state, placing the local program in send state. The local TPF transaction program can now call `tppc_send_data`.

### RECEIVE\_WHATRCV\_CONFIRMDLLOC

This value indicates that the remote program called `tppc_deallocate` with **type=DEALLOCATE\_TYPE\_CONFIRM**. The local TPF transaction program can respond by issuing `tppc_confirmed` or `tppc_send_error`.

### rtsrcvd

This is a pointer to a 1-byte field that contains the enumeration type `t_rtsrcvd`, where the `REQUEST_TO_SEND_RECEIVED` indication is placed. The value of the `REQUEST_TO_SEND_RECEIVED` indication is **RTSRCVD\_YES**, indicating that a `REQUEST_TO_SEND` indication was received from the remote transaction program. The remote program called `tppc_request_to_send` requesting the local TPF transaction program to enter receive state and placing the remote transaction program in send state.

**fill** This argument must belong to the enumeration type `t_receive_fill`, defined in `tppc.h`. Use the value **RECEIVE\_FILL\_LL**, which specifies that the program is to receive 1 complete or truncated logical record, or a portion of a logical record that is equal to the length specified by the **length** parameter. See the programming considerations for an explanation of the record format.

**Note:** The TPF system does not support the `BUFFER` option defined by the LU 6.2 architecture.

### wait

This argument must belong to the enumeration type `t_receive_wait`, defined in `tppc.h`. Use the value **RECEIVE\_WAIT\_YES**, which performs the function of `RECEIVE_AND_WAIT`.

**Note:** The TPF system does not support the `RECEIVE_IMMEDIATE` function defined by the LU 6.2 architecture.

### data

This is a pointer to the location in memory that will receive the data. It can also be a `NULL` pointer, which indicates the data is to be placed in a block on data level 0 in AMOSG format.<sup>1</sup>

### length

This is a pointer to a 2-byte field that specifies the maximum length of data the program can receive. The value can be from 0–32 767. The value can also be a `NULL` pointer but only if **data** is also `NULL`.<sup>1</sup> If both arguments are `NULL`, the length of the data received is determined by the logical record length.

When control is returned to the transaction program, the value is the actual amount of data the program received up to the maximum. If the program receives information other than data, the value remains unchanged.

---

1. If either **data** or **length** is `NULL` while the other parameter is not `NULL`, a parameter check is issued.

## Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the tppc\_receive function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_PK_BAD_OPTION	....	00C62074
LU62RC_INVALID_LENGTH	....	00000006
LU62RC_STATE_CHECK	0002	
LU62RC_SKRECEV_BADSTATE	....	000000B1
LU62RC_ALLOC_ERROR	0003	
LU62RC_SECURITY_NOT_VALID	....	080F6051
LU62RC_TP_NOT_AVAIL_RETRY	....	084B6031
LU62RC_TP_NOT_AVAIL_NO_RETRY	....	084C0000
LU62RC_TPN_NOT_RECOGNIZED	....	10086021
LU62RC_PIP_NOT_SPECIFIED_CORRECTLY	....	10086032
LU62RC_CONV_TYPE_MISMATCH	....	10086034
LU62RC_SYNLVL_NOTSUPORT	....	10086041
LU62RC_DLLOC_ABEND_PGM	0006	
LU62RC_DLLOC_ABEND_SVC	0007	
LU62RC_DLLOC_ABEND_TMR	0008	
LU62RC_DLLOC_NORMAL	0009	
LU62RC_PGMERR_NOTRUNC	000C	
LU62RC_PGMERR_TRUNC	000D	
LU62RC_PGMERR_PURGING	000E	
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NORETRY	0010	
LU62RC_SVCERR_NOTRUNC	0011	
LU62RC_SVCERR_TRUNC	0012	
LU62RC_SVCERR_PURGING	0013	
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The conversation must be in send or receive state.
- The value supplied in **resid** must be the resource ID returned by the tppc\_allocate function or one that was assigned by an incoming ATTACH.
- The transaction program must check both the return code and the WHAT\_RECEIVED (**whatrcv**) indicator.

## tppc\_receive

**Note:** If the return code is not LU62RC\_OK, **whatrcv** has no meaning and should not be examined.

- If the value for **length** and **data** is NULL and the WHAT\_RECEIVED indicator indicates that data is returned to the transaction program, one of the following occurred:
  - The program received a complete logical record or the last remaining portion of a complete logical record.<sup>2</sup> The WHAT\_RECEIVED indicator is set to **RECEIVE\_WHATRCV\_DATACOMPLETE**.
  - The program receives an incomplete logical record because it was truncated.<sup>3</sup>

The data is presented in a 4KB working storage block on data level 0 in AMSG format (defined by the AM0SG DSECT).

If an entire logical record or the beginning of a truncated record is received, the complete logical record length is placed in the first 2 bytes of the logical record (AM0TXT), followed by the text of the record.

If the last remaining portion of a logical record is received, the text of the record is not preceded by a logical length field and begins at AM0TXT.

If the complete logical record does not fit in a single block, the remainder of the logical record is chained in 4KB, short-term file pool records. The standard TPF forward chain field (AM0FWD) contains the file address of the next segment of a logical record. A forward chain field containing zeros denotes the end of the chain. The standard TPF message length field (AM0CCT) contains the number of bytes of text in each of the segments.<sup>4</sup>

If the entire logical record (or the beginning of a truncated record) is received, the logical record length presented in the first 2 bytes of a logical record is the length of the complete logical record, while the contents of the AM0CCT field contains the physical length of each segment of a logical record. If the completion of a logical record is received, the sum of the AM0CCT fields of the segment indicates the total amount of data received.

- If you specify the **data** and **length** parameters and the WHAT\_RECEIVED indicator indicates that data is returned to the transaction program, one of the following occurred:
  - The program received a complete logical record or the last remaining portion of a complete logical record. The length of the record or portion of the record is equal to or less than the length specified on the **length** parameter. If the length received is less than the value pointed to by the **length** parameter, the value is updated to indicate the actual amount of data received. The WHAT\_RECEIVED indicator is set to **RECEIVE\_WHATRCV\_DATACOMPLETE**.
  - The program receives an incomplete logical record, which can occur because:
    - The length of the logical record is greater than the length specified by the **length** parameter; in this case, the amount received equals the length specified.

---

2. The last remaining portion of a complete logical record can be returned if this RECEIVE followed a RECEIVE that used the **data** and **length** parameters and had a WHAT\_RECEIVED indicator of **RECEIVE\_WHATRCV\_DATAINCOMPLETE**.

3. Truncated logical records can be received if the remote transaction program caused data truncation by issuing tppc\_send\_error or tppc\_deallocate **type=** any ABEND.

4. The AM0CCT field is equal to the actual text count in this block plus 5, which is the length of the AM0SG filler. Thus the physical length of data in a block is (AM0CCT - 5).

- Only a portion of the logical record is available because it was truncated, the portion being equal to or less than the length specified by the **length** parameter.<sup>3</sup>

The WHAT\_RECEIVED indicator is set to

**RECEIVE\_WHATRCV\_DATAINCOMPLETE**. The program must issue another tppc\_receive (or possibly multiple tppc\_receives) to receive the remainder of the logical record.

**Note:** A **length** parameter value of zero has no special significance. The type of information available is indicated by the **rcode** parameter and the **whatrcv** parameters, as usual. If data is available, the **whatrcv** parameter contains **RECEIVE\_WHATRCV\_DATAINCOMPLETE**, but the program receives no data.

- The maximum size of a logical record is 32 767.
- On return, the value pointed to by the **length** parameter may have been changed to show the amount of data actually received.
- If there is no data available, TPF's EVENT and POST facility is used to suspend the ECB until data arrives. Because the ECB is suspended, all unnecessary resources should be released before this verb is issued. Failure to do so can cause serious system degradation.

**Note:** You can avoid the problem of suspended ECBs by using the tppc\_activate\_on\_receipt function instead of tppc\_receive. See "tppc\_activate\_on\_receipt—Activate a Program after Information Received" on page 741 for more information.

- If the program does not receive any data in a certain amount of time, TPF/APPC support issues a tppc\_deallocate with **DEALLOCATE\_TYPE\_ABENDP** to terminate the conversation. The amount of time that the system waits is determined by the value you specify for the TPRECV parameter on the SNAKEY macro. See *TPF ACF/SNA Network Generation* for information about the SNAKEY macro.
- See "Programming Considerations for Basic Conversation Functions" on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example waits for information to arrive on the specified conversation and then receives the information.

```
#include <tppc.h>

    unsigned int      resource_id;
    struct tppc_return_codes return_code;
    enum t_receive_whatrcv what_received;
    enum t_rtsrcvd      request_received;
    unsigned short int  length;
    unsigned char       *data;
    :
    :
/* set up resource_id with the value returned from the allocate verb */
    :
    :
tppc_receive(&resource_id,&return_code,&what_received,&request_received, \
    RECEIVE_FILL_LL,RECEIVE_WAIT_YES,data,&length);
```

**tppc\_receive**

```

                                /* normal processing path          */
                                :

```

## Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_allocate—Allocate a Conversation” on page 745.

## tppc\_request\_to\_send—Request Change to Send State

This function notifies the remote transaction program that the local TPF transaction program is requesting to enter send state for the conversation. The conversation changes to send state when the local TPF program subsequently receives a SEND indication from the remote program.

### Format

```
#include <tppc.h>
void tppc_request_to_send(unsigned int *resid,
                        struct tppc_return_codes *rcode);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

### Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the tppc\_request\_to\_send function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_STATE_CHECK	0002	
LU62RC_SKRTSND_BADSTATE	....	000000E1
LU62RC_TPF_ABEND	FFFF	

### Programming Considerations

- The conversation must be in one of the following states:
    - send
    - receive
    - received-confirm
    - received-confirm-send
    - received-confirm-deallocate
  - The value supplied in **resid** must be the resource ID returned by the tppc\_allocate function or one that was assigned by an incoming ATTACH.
  - After issuing the tppc\_request\_to\_send function, the local TPF transaction program must call the tppc\_receive function to wait for the SEND indication to arrive from the remote program. The SEND indication is received in the WHAT\_RECEIVED (**whatrcv**) parameter of the tppc\_receive function. (See “tppc\_receive—Receive Information” on page 771.)
- For the local program to receive permission to send data, the remote program must do one of the following:

## tppc\_request\_to\_send

- Issue a RECEIVE verb
- Issue a PREPARE\_TO\_RECEIVE verb
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example notifies the remote transaction program that the local transaction program is requesting to enter send state for the conversation.

```
#include <tppc.h>

        unsigned int      resource_id;
        struct tppc_return_codes return_code;
        :
        :
/* set up resource_id with the value returned from the allocate verb */
        :
        :
tppc_request_to_send(&resource_id,&return_code);
                        /* normal processing path          */
        :
        :
```

## Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_allocate—Allocate a Conversation” on page 745.



## tppc\_send\_data–Send Data to Remote Transaction Program

This function sends data to the remote transaction program. The data sent consists of logical records.

### Format

```
#include <tppc.h>
void tppc_send_data(unsigned int *resid,
                    struct tppc_return_codes *rcode,
                    enum t_rtsrcvd *rtsrcvd,
                    unsigned char *data,
                    unsigned short int *length);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

#### rtsrcvd

This is a pointer to a 1-byte field that contains the enumeration type t\_rtsrcvd, where the REQUEST\_TO\_SEND\_RECEIVED indication is placed. The REQUEST\_TO\_SEND\_RECEIVED indication is:

##### RTSRCVD\_YES

This value indicates that a REQUEST\_TO\_SEND indication was received from the remote transaction program. The remote program called tppc\_request\_to\_send, requesting the local TPF transaction program to enter receive state and placing the remote transaction program in send state.

Any other value indicates that a REQUEST\_TO\_SEND notification was not received.

#### data

This is a pointer to the location in memory that contains the data to be sent.<sup>5</sup>

#### length

This is a pointer to a 2-byte field that contains the length of data to be sent. The value can be from 0–32 767. The value can also be a NULL pointer, but only if **data** is also NULL.<sup>5</sup>

The data length is not related in any way to the length of the logical record. It is used only to determine the length of the data located at the address specified by the **data** parameter.

**Note:** If **data** and **length** are NULL, the data is assumed to be on the data level 0 in AM0SG format. See the programming considerations for an explanation of the record format.

### Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the tppc\_send\_data function. A complete list of the return codes and their definitions can be found in Table 37 on page 731

5. If either **data** or **length** is NULL while the other parameter is not NULL, a parameter check is issued.

and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_INVALID_LENGTH	....	00000006
LU62RC_PKSENDD_BADLL	....	000000F1
LU62RC_STATE_CHECK	0002	
LU62RC_SKSENDD_BADSTATE	....	000000F2
LU62RC_ALLOC_ERROR	0003	
LU62RC_SECURITY_NOT_VALID	....	080F6051
LU62RC_TP_NOT_AVAIL_RETRY	....	084B6031
LU62RC_TP_NOT_AVAIL_NO_RETRY	....	084C0000
LU62RC_TPN_NOT_RECOGNIZED	....	10086021
LU62RC_PIP_NOT_SPECIFIED_CORRECTLY	....	10086032
LU62RC_CONV_TYPE_MISMATCH	....	10086034
LU62RC_SYNLVL_NOTSUPPORT	....	10086041
LU62RC_DLLOC_ABEND_PGM	0006	
LU62RC_DLLOC_ABEND_SVC	0007	
LU62RC_DLLOC_ABEND_TMR	0008	
LU62RC_PGMERR_PURGING	000E	
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NORETRY	0010	
LU62RC_SVCERR_PURGING	0013	
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The conversation must be in `send` state.
- The value supplied in **resid** must be the resource ID returned by the `tppc_allocate` function or one that was assigned by an incoming ATTACH.
- The following considerations apply if **data** and **length** are NULL:
  - The data to be sent must be in the main storage block attached to data level 0 (D0) in AMMSG format. The block can be a small (381), large (1055), or 4KB storage block. The data must be in logical record format; that is, the first 2 bytes contain the length of the logical record followed by the data. The storage block can contain 1 or more full logical records or it can contain a partial logical record. The number of bytes in the block to be sent is always in the AMMSG length field (AM0CCT) regardless of the logical record boundary. It is the transaction program's responsibility to insure the accuracy of both the logical record length field (LL preceding the text) and the storage block length field (AM0CCT).<sup>6</sup>

6. The AM0CCT field is equal to the actual text count in this block plus 5, which is the length of the AM0SG filler. Thus the physical length of data in a block is (AM0CCT - 5).

- If the complete logical record does not fit into 1 block, the message can be forward chained. The number of bytes of text for each of the chained segments is placed in the AMSG length field (AM0CCT), and the file address of the next segment is placed in the standard TPF forward chain field (AM0FWD). A forward chain field containing zeros indicates the last segment of a chained logical record. The logical record length in the first 2 bytes of the first segment contains the length of the entire logical record, and the AM0CCT field of each segment contains the physical number of bytes of text in each segment.
- The transaction program can also send logical records by issuing multiple `tppc_send_data` functions for segments of the logical record. In this case, the first segment must contain the logical record length (LL) preceding the first byte of the text, and the AMSG length field (AM0CCT) must contain the number of bytes in each storage block.
- If the return code is not LU62RC\_OK, the data remains on D0 unchanged.
- The following considerations apply if **data** and **length** are not NULL:
  - The data must be in logical record format; that is, the first 2 bytes of the logical record contain the length of the record followed by the data.
  - The number of bytes to be sent is always the value pointed to by the **length** parameter regardless of the logical record boundary.
  - It is the transaction program's responsibility to insure the accuracy of both the logical record length field (LL preceding the text) and the **length** parameter.
  - The entire storage area referred to by the **data** and **length** parameters must be addressable by the TPF/APPC support code.
- The maximum size of a logical record is 32 767.
- When the **rtsrcvd** parameter is **RTSRCVD\_YES**, the remote program is requesting the local TPF transaction program to enter receive state and thereby place the remote program in send state. The local TPF transaction program enters receive state by issuing `tppc_receive` or `tppc_prepare_to_receive`. The remote partner program enters the corresponding send state when it calls `tppc_receive` and receives the SEND indicator on the **whatrcv** parameter.
- See to "Programming Considerations for Basic Conversation Functions" on page 735 for programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example sends data to the remote transaction program.

```
#include <tppc.h>

    unsigned int      resource_id;
    struct tppc_return_codes return_code;
    enum t_rtsrcvd    request_received;
    unsigned short int length;
    unsigned char     *data;
    :
    :
/* set up resource_id with the value returned from the allocate verb */
    :
    :
tppc_send_data(&resource_id,&return_code,&request_received, \
              data,&length);
    :
    :
/* normal processing path */
    :
    :
```

## Related Information

- "Return Codes for Basic Conversation Functions" on page 731
- "tppc\_allocate—Allocate a Conversation" on page 745.

## tppc\_send\_error–Send Error Notification

This function informs the remote transaction program that the local TPF transaction program detected an error.

Upon successful completion of this function, the local TPF transaction program is in send state, and the remote program is in receive state. Further action is defined by the transaction program's logic.

### Format

```
#include <tppc.h>
void tppc_send_error(unsigned int *resid,
                    struct tppc_return_codes *rcode,
                    enum t_rtsrcvd *rtsrcvd,
                    enum t_send_error_type type
                    enum t_send_error_logdata logdata);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

#### rtsrcvd

This argument is a pointer to a 1-byte field that contains the enumeration type t\_rtsrcvd, where the REQUEST\_TO\_SEND\_RECEIVED indication is placed. This indication contains **RTSRCVD\_YES**, which indicates a REQUEST\_TO\_SEND indication was received from the remote transaction program. The remote program called tppc\_request\_to\_send requesting the local TPF transaction program to enter receive state and placing the remote transaction program in send state.

#### type

This specifies the type of error indication to be sent. This argument must belong to the enumeration type t\_send\_error\_type, defined in tppc.h. Use one of the following values:

##### **SEND\_ERROR\_TYPE\_PROG**

This value indicates that the TPF transaction program detected an error that is to be reported to the remote transaction program.

##### **SEND\_ERROR\_TYPE\_SVC**

This value indicates that a TPF service program detected an error that is to be reported to the remote LU service program.

#### logdata

This specifies whether the error indication is logged. This argument must belong to the enumeration type t\_send\_error\_logdata, defined in tppc.h. Use the value **SEND\_ERROR\_LOGDATA\_NO**.

**Note:** The TPF system does not support the YES option defined by the LU 6.2 architecture.

### Return Codes

The following tables contain a list of the primary and secondary return codes that can be returned to the program that called the tppc\_send\_error function based on

the conversation state. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

*Table 39. Return Codes When Conversation in SEND State*

<b>Symbolic Name</b>	<b>Primary Code</b>	<b>Secondary Code</b>
LU62RC_OK	0000	
LU62RC_POSTED_DATA	....	00000000
LU62RC_POSTED_NODATA	....	00000001
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_PK_BAD_OPTION	....	00C62074
LU62RC_STATE_CHECK	0002	
LU62RC_SKSENDE_BADSTATE	....	00000111
LU62RC_ALLOC_ERROR	0003	
LU62RC_SECURITY_NOT_VALID	....	080F6051
LU62RC_TP_NOT_AVAIL_RETRY	....	084B6031
LU62RC_TP_NOT_AVAIL_NO_RETRY	....	084C0000
LU62RC_TPN_NOT_RECOGNIZED	....	10086021
LU62RC_PIP_NOT_SPECIFIED_CORRECTLY	....	10086032
LU62RC_CONV_TYPE_MISMATCH	....	10086034
LU62RC_SYNLVL_NOTSUPPORT	....	10086041
LU62RC_DLLOC_ABEND_PGM	0006	
LU62RC_DLLOC_ABEND_SVC	0007	
LU62RC_DLLOC_ABEND_TMR	0008	
LU62RC_PGMERR_PURGING	000E	
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NORETRY	0010	
LU62RC_SVCERR_PURGING	0013	
LU62RC_TPF_ABEND	FFFF	

*Table 40. Return Codes When Conversation in RECEIVE State*

<b>Symbolic Name</b>	<b>Primary Code</b>	<b>Secondary Code</b>
LU62RC_OK	0000	
LU62RC_DLLOC_NORMAL	0009	
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NORETRY	0010	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_PK_BAD_OPTION	....	00C62074
LU62RC_TPF_ABEND	FFFF	

Table 41. Return Codes When Conversation in CONFIRM State.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_PK_BAD_OPTION	....	00C62074
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NORETRY	0010	
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The conversation must be in one of the following states:  
send  
receive  
received-confirm  
received-confirm-send  
received-confirm-deallocate.
- The value supplied in **resid** must be the resource ID returned by the `tppc_allocate` function or one that was assigned by an incoming ATTACH.
- The transaction programs can use this function for various application level functions. For example, the program can call this function to truncate an incomplete logical record, to inform the remote program of an error it detected in data it received, or to reject a confirmation request.
- If the local TPF transaction program calls `tppc_send_error` while in receive state, purging of all buffered input occurs. The incoming information that is purged includes both data (logical records) and confirmation requests. In addition, if the confirmation request was due to the remote transaction program's calling `tppc_deallocate` with **type=DEALLOCATE\_TYPE\_CONFIRM**, the DEALLOCATE request is also purged. After issuing the `tppc_send_error` function from receive state, the local transaction program is in send state, and the remote transaction program is in receive state.
- When the **rtsrcvd** parameter is **RTSRCVD\_YES**, the remote program is requesting the local TPF transaction program to enter receive state and thereby place the remote program in send state. The local TPF transaction program enters receive state by issuing `tppc_receive` or `tppc_prepare_to_receive`. The remote partner program enters the corresponding send state when it calls `tppc_receive` and receives the SEND indicator on the **whatrcv** parameter.
- This verb resets or cancels posting. If posting is active and the conversation was posted, or if posting is active and the conversation was not posted, posting is cancelled.
- The truncation implied by the use of this verb can cause any of the following to be truncated:
  - Data, sent by means of the `tppc_send_data` function.
  - Confirmation requests, sent by means of `tppc_confirm`, `tppc_prepare_to_receive`, or `tppc_deallocate`.
  - Deallocation request sent by mean of `tppc_deallocate` with **type=DEALLOCATE\_TYPE\_CONFIRM**.

- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example informs the remote transaction program that the local TPF transaction program detected an error.

```
#include <tppc.h>

    unsigned int      resource_id;
    struct tppc_return_codes return_code;
    enum t_rtsrcvd     request_received;
    :
    :
/* set up resource_id with the value returned from the allocate verb */
    :
    :
tppc_send_error(&resource_id,&return_code,&request_received,      \
                SEND_ERROR_TYPE_PROG,SEND_ERROR_LOGDATA_NO);
    :                               /* normal processing path      */
    :
```

## Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_allocate—Allocate a Conversation” on page 745.

## tppc\_test–Test Conversation

This function tests the conversation for a specified condition. The results of the test are indicated in the return code.

### Format

```
#include <tppc.h>
void tppc_test(unsigned int *resid,
               struct tppc_return_codes *rcode,
               enum t_test_test test);
```

#### resid

This is a pointer to a 4-byte field that contains the resource ID. This resource ID must be the ID assigned on the initial ALLOCATE for this conversation or one that was assigned by an incoming ATTACH.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

#### test

This specifies the type of test to be performed. This argument must belong to the enumeration type t\_test\_test, defined in tppc.h. Use one of the following values:

#### TEST\_TEST\_POSTED

This value specifies to test whether the conversation was posted.

#### TEST\_TEST\_RT SRCVD

This value specifies to test whether the REQUEST\_TO\_SEND notification was received from the remote transaction program.

## Return Codes

The following tables contain a list of the primary and secondary return codes that can be returned to the program that called the tppc\_test function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Table 42. Return Codes for TEST=POSTED

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_POSTED_DATA	....	00000000
LU62RC_POSTED_NOTDATA	....	00000001
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_STATE_CHECK	0002	
LU62RC_NOT_RCV_STATE	....	00000122
LU62RC_ALLOC_ERROR	0003	
LU62RC_SECURITY_NOT_VALID	....	080F6051
LU62RC_TP_NOT_AVAIL_RETRY	....	084B6031
LU62RC_TP_NOT_AVAIL_NO_RETRY	....	084C0000
LU62RC_TPN_NOT_RECOGNIZED	....	10086021



Table 42. Return Codes for TEST=POSTED (continued)

Symbolic Name	Primary Code	Secondary Code
LU62RC_PIP_NOT_SPECIFIED_CORRECTLY	....	10086032
LU62RC_CONV_TYPE_MISMATCH	....	10086034
LU62RC_SYNLVL_NOTSUPPORT	....	10086041
LU62RC_DLLOC_ABEND_PGM	0006	
LU62RC_DLLOC_ABEND_SVC	0007	
LU62RC_DLLOC_ABEND_TMR	0008	
LU62RC_DLLOC_NORMAL	0009	
LU62RC_POSTING_NOTACTIV	000B	
LU62RC_PGMERR_NOTRUNC	000C	
LU62RC_PGMERR_TRUNC	000D	
LU62RC_PGMERR_PURGING	000E	
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NORETRY	0010	
LU62RC_SVCERR_NOTRUNC	0011	
LU62RC_SVCERR_TRUNC	0012	
LU62RC_SVCERR_PURGING	0013	
LU62RC_LLRCV_UNSUCESSFUL	0014	
LU62RC_TPF_ABEND	FFFF	

Table 43. Return Codes for TEST=RTSRCVD

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_LLRCV_UNSUCESSFUL	0014	
LU62RC_TPF_ABEND	FFFF	

Table 44. Return code when TEST is neither POSTED nor RTSRCVD

Symbolic name	Primary Code	Secondary Code
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_OPTION	....	00C62074

## Programming Considerations

- The conversation must be in one of the following states, depending on the type of test requested:

**Type**                      **State Required**

**POSTED**                  receive

**RTSTRCVD**              send or receive.

## tppc\_test

- The value supplied in **resid** must be the resource ID returned by the `tppc_allocate` function or one that was assigned by an incoming ATTACH.
- The **test=TEST\_TEST\_POSTED** option is valid only when the conversation is in receive state.
- The **test=TEST\_TEST\_POSTED** option is used in conjunction with the `tppc_post_on_receipt` function. The use of `tppc_post_on_receipt` and `tppc_test` allows a program to continue its processing while waiting for information to become available. The TPF transaction program calls `tppc_post_on_receipt` for one or more conversations and then calls `tppc_test` for each conversation to determine when information is available to be received.
- For the **test=TEST\_TEST\_POSTED** option, the return code indicates whether posting is active, whether the conversation was posted, and whether the information available is data or not. The TPF transaction program must call `tppc_receive` to receive the information for a conversation that was posted.
- The **test=TEST\_TEST\_POSTED** option returns LU62RC\_LLRCV\_UNSUCESSFUL if the conversation was not posted.
- Posting is active for a conversation on which the `tppc_post_on_receipt` function was called and posting was not reset or canceled. See “`tppc_post_on_receipt`–Set Posting Active for a Conversation” on page 765 for more information on posting.
- The **test=TEST\_TEST\_RTSRVCD** option returns LU62RC\_OK if the REQUEST\_TO\_SEND was received. Otherwise, the REQUEST\_TO\_SEND indication was not received from the remote transaction program and, LU62RC\_LLRCV\_UNSUCESSFUL is returned.
- To enter receive state, the local transaction program must call the appropriate function, such as `tppc_receive` or `tppc_prepare_to_receive`, when the REQUEST\_TO\_SEND indication is received.
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example tests if a conversation was posted.

```
#include <tppc.h>

    unsigned int      resource_id;
    struct tppc_return_codes return_code;
    :
/* set up resource_id with the value returned from the allocate verb */
    :
tppc_test(&resource_id,&return_code,TEST_TEST_POSTED);
    :                               /* normal processing path          */
    :
```

## Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “`tppc_allocate`–Allocate a Conversation” on page 745
- “`tppc_post_on_receipt`–Set Posting Active for a Conversation” on page 765.

## tppc\_wait–Wait for Posting

This function waits for posting to occur on any mapped or basic conversation from a list of conversations. Posting of a conversation occurs when posting is active for the conversation and information that the conversation can receive, such as data, conversation status, or a request for confirmation, is available.

### Format

```
#include <tppc.h>
void tppc_wait(unsigned int *residl,
               struct tppc_return_codes *rcode,
               unsigned int *respstd);
```

#### residl

This is a pointer to a variable-length field that contains the resource IDs of the conversations for which posting is expected. The resource IDs must be the resource IDs assigned on the initial ALLOCATE for the conversations or assigned by an incoming ATTACH. The variable length field is in the format:

- Resource ID, which is 4 bytes repeated for each conversation.
- Delimiter, which is 4 bytes of binary zeros.

**Note:** This list can contain as many as 20 entries, which allows space for 19 resource IDs and the delimiter.

#### rcode

This is a pointer to the structure tppc\_return\_codes, defined in tppc.h, where the return code is to be placed.

#### respstd

This is a pointer to a 4-byte field where the resource ID of a posted conversation is returned.

## Return Codes

The following table contains a list of the primary and secondary return codes that can be returned to the program that called the tppc\_wait function. A complete list of the return codes and their definitions can be found in Table 37 on page 731 and Table 38 on page 732.

Symbolic Name	Primary Code	Secondary Code
LU62RC_OK	0000	
LU62RC_POSTED_DATA	....	00000000
LU62RC_POSTED_NOTDATA	....	00000001
LU62RC_PARAMETER_CHECK	0001	
LU62RC_PK_BAD_TCBID	....	00000001
LU62RC_PK_BAD_CONVID	....	00000002
LU62RC_PK_EMPTY_RESIDL	....	00000003
LU62RC_STATE_CHECK	0002	
LU62RC_NOT_RCV_STATE	....	00000122
LU62RC_ALLOC_ERROR	0003	
LU62RC_SECURITY_NOT_VALID	....	080F6051
LU62RC_TP_NOT_AVAIL_RETRY	....	084B6031
LU62RC_TP_NOT_AVAIL_NO_RETRY	....	084C0000

Symbolic Name	Primary Code	Secondary Code
LU62RC_TPN_NOT_RECOGNIZED	....	10086021
LU62RC_PIP_NOT_SPECIFIED_CORRECTLY	....	10086032
LU62RC_CONV_TYPE_MISMATCH	....	10086034
LU62RC_SYNLVL_NOTSUPPORT	....	10086041
LU62RC_DLLOC_ABEND_PGM	0006	
LU62RC_DLLOC_ABEND_SVC	0007	
LU62RC_DLLOC_ABEND_TMR	0008	
LU62RC_DLLOC_NORMAL	0009	
LU62RC_POSTING_NOTACTIV	000B	
LU62RC_PGMERR_NOTRUNC	000C	
LU62RC_PGMERR_TRUNC	000D	
LU62RC_PGMERR_PURGING	000E	
LU62RC_CONVFAIL_RETRY	000F	
LU62RC_CONVFAIL_NORETRY	0010	
LU62RC_SVCERR_NOTRUNC	0011	
LU62RC_SVCERR_TRUNC	0012	
LU62RC_SVCERR_PURGING	0013	
LU62RC_TPF_ABEND	FFFF	

## Programming Considerations

- The conversation must be in receive state.
- The values supplied in **residl** must be the resource IDs returned by the `tppc_allocate` function or IDs that were assigned by an incoming ATTACH.
- This function is intended to be used in conjunction with the `tppc_post_on_receipt` function. The use of these functions allows a program to perform synchronous receiving from multiple conversations. The program calls `tppc_post_on_receipt` for each conversation and then calls this function for all the conversations in the resource list. The WAIT is satisfied when any conversation in the list has information available to be received.
- The value returned in **respstd** is one of the resource IDs specified in **residl**. The resource ID returned identifies the conversation that was posted.
- The return code indicates the posting state for the conversation identified in the **respstd** parameter. A secondary return code of LU62RC\_POSTED\_DATA indicates that a data message is available for receipt. A secondary return code of LU62RC\_POSTED\_NOTDATA indicates that information other than a data message is available, such as a confirmation request of a SEND indication.
- Posting is reset for the conversation returned in the **respstd** parameter. The TPF transaction program must call the `tppc_receive` function in order to receive the available information.
- Posting may or may not be active for the resource IDs specified in **residl**. This function waits for posting to occur on any conversation's resource ID in the list that has posting active.
- This function uses TPF's EVENT and POST facility to suspend the ECB until posting occurs on one of the conversations. Since the ECB is suspended, all

unnecessary resources should be released before calling this verb. Failure to do so can cause serious system performance degradation.

- If posting does not occur within a certain amount of time, TPF/APPC support issues a tppc\_deallocate with **DEALLOCATE\_TYPE\_ABENDP** to terminate all the conversations in the list. The amount of time that the system waits is determined by the value you specify for the TPWAIT parameter on the SNAKEY macro. See *TPF ACF/SNA Network Generation* for information about the SNAKEY macro.
- See “Programming Considerations for Basic Conversation Functions” on page 735 for additional programming considerations relating to the TPF/APPC basic conversation functions.

## Examples

The following example waits for posting to occur on any one of a list of conversations.

```
#include <tppc.h>

        unsigned int      resource_id_list[10];
        struct tppc_return_codes return_code;
        unsigned int      posted_resource_id;
        :
resource_id_list[0] = conversation_1;
resource_id_list[1] = conversation_2;
resource_id_list[2] = conversation_3;
resource_id_list[3] = 0;

/* zero resource id indicates the end of the list.          */
:
tppc_wait(resource_id_list,&return_code,&posted_resource_id);
/* normal processing path                                  */
:

```

## Related Information

- “Return Codes for Basic Conversation Functions” on page 731
- “tppc\_allocate—Allocate a Conversation” on page 745
- “tppc\_post\_on\_receipt—Set Posting Active for a Conversation” on page 765.

**tppc\_wait**

---

## TPF/APPC Mapped Conversation Functions

The function provided by the TPF Advanced Program-to-Program Communications (TPF/APPC) interface is an implementation of IBM's Advanced Program-to-Program Communications (APPC) architecture. TPF/APPC is an interface that allows TPF transaction programs to communicate with remote SNA nodes that have implemented the APPC interface using LU 6.2 protocols. See the *TPF ACF/SNA Data Communications Reference* for more information about TPF/APPC support in general. Before using this support, you must be familiar with the SNA LU 6.2 protocol architecture as defined in the following documents:

- *IBM Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2*
- *IBM Systems Network Architecture LU 6.2 Reference: Peer Protocols*
- *IBM Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2.*

This chapter contains a general description of the TPF/APPC mapped conversation interface, and an alphabetic listing of each function. The following information is shown for each function:

**Format** The function prototype and a description of any parameters.

**Description** What service the function provides.

**Return Codes** A list of return codes for the specific function.

**Note:** This section replaces the Normal Return and Error Return sections that are used in "TPF API Functions" on page 1.

### Programming Considerations

Remarks that help the programmer to understand the correct use of the function and any side effects that may occur when the function is executed. If the use of a particular function affects the use of another function, that is also described.

**Example** A code segment showing a sample function call.

### Related Functions

Where to find additional information pertinent to this function.

### Notes:

1. Remember to include `tpfeq.h` in the source program, even though it is not shown in any of the function prototypes. The `tpfeq.h` header must be the first header file included in any TPF C source module.
2. Programs that issue any of the API functions for the mapped conversation verbs must be authorized in the allocator to issue restricted use macros and must have KEY0 write authorization.

See "TPF/APPC Basic Conversation Functions" on page 723. for descriptions of the TPF/APPC basic conversation functions.

---

## TPF/APPC Mapped Conversation Interface Overview

The TPF/APPC mapped interface is based on the communication element of the SAA's Common Programming Interface (CPI). Although TPF/APPC does not fully conform to CPI Communications, standard CPI Communications programs can be converted easily, provided the programs do not use the features described below

that TPF does not support. See the *Systems Application Architecture Common Programming Interface Communications Reference* for more information about CPI Communications.

The TPF/APPC mapped interface differs from CPI Communications in the following:

- Sync point is not supported.
- Basic conversations are not supported.
- Log data is not supported.
- The *receive\_type* characteristic cannot be **CM\_RECEIVE\_IMMEDIATE**.
- Conversations that are still active after the controlling transaction program exits are not automatically deallocated.
- FLUSH has additional return codes.

As a result of these changes, some CPI Communications functions and characteristics no longer have meaning or can have only one value. Therefore, the following functions are not supported:

- *cmect* (Extract\_Conversation\_Type)
- *cmsct* (Set\_Conversation\_Type)
- *cmsf* (Set\_Fill)
- *cmsld* (Set\_Log\_Data)
- *cmsrt* (Set\_Receive\_Type).

The following characteristics are not supported:

- *conversation\_type*
- *fill*
- *log\_data*
- *log\_data\_length*
- *receive\_type*.

## Characteristics

Characteristics are values that are stored internally and keep the transaction program from having to specify the same values on function calls over and over. There are special functions to change and examine the values of the various characteristics. The characteristics, which are shown in *italics* throughout this chapter, are:

- *deallocate\_type*
- *error\_direction*
- *mode\_name*
- *mode\_name\_length*
- *partner\_LU\_name*
- *partner\_LU\_name\_length*
- *prepare\_to\_receive\_type*
- *return\_control*
- *send\_type*
- *sync\_level*
- *TP\_name*
- *TP\_name\_length*.

The set functions provide details about these characteristics. In addition, the programming considerations for each function list any characteristics that are affected by that function.

## Conversation States

The following is a list of the conversation states for the TPF/APPC mapped conversations:



<u>reset</u>	There is no conversation.
<u>initialize</u>	The conversation is initialized but not allocated.
<u>send</u>	The program can send data.
<u>send-pending</u>	The cmrcv (RECEIVE) function returned <b>data_received</b> with the value <b>CM_COMPLETE_DATA_RECEIVED</b> and <b>status_received</b> with the value <b>CM_SEND_RECEIVED</b> . When a conversation is in this state, the <i>error_direction</i> characteristic affects how an error is reported to the remote program. (See “cmsed–Set the Error_Direction Characteristic” on page 833.)
<u>receive</u>	The program can receive data.
<u>confirm</u>	The transaction program received a confirmation request from the remote transaction program.
<u>confirm-send</u>	The remote transaction program sent a cmpttr (PREPARE_TO_RECEIVE) request with <i>prepare_to_receive_type</i> set to <b>CM_PREP_TO_RECEIVE_CONFIRM</b> , or with <i>prepare_to_receive_type</i> set to <b>CM_PREP_TO_RECEIVE_SYNC_LEVEL</b> and <i>sync_level</i> set to <b>CM_CONFIRM</b> . If the local transaction program does a cmcfmd (CONFIRMED), the PREPARE_TO_RECEIVE takes effect.
<u>confirm-deallocate</u>	The transaction program received a confirmation request and a deallocation notification from the remote program. If the local program replies with a cmcfmd (CONFIRMED), the deallocation takes effect.

**Note:** Because the TPF system does not support sync point, the following states defined by CPI Communications are not used:

- defer-receive
- defer-deallocate
- sync-point
- sync-point-send
- sync-point-deallocate.

## Side Information Table

The side information table holds combinations of partner LU names, remote transaction program names, and mode names. Each combination is identified by a unique symbolic destination name. The side information table offline program (CHQI) provides a way of generating the side information table, verifying the syntax of the entries before bringing them online, and managing the data contained in the online information table. You can also build or modify the table online using the ZNSID command. See the *TPF ACF/SNA Data Communications Reference* for more information about the CHQI program. See *TPF Operations* for more information about ZNSID.

## Return Codes for Mapped Conversation Functions

Each TPF/APPC mapped function returns a 4-byte return code. Table 45 contains a list of all the return codes and their definitions. Not all return codes are possible for every function. The individual TPF/APPC mapped function descriptions document valid return codes for that function. The symbolic names of the return code values are defined in the `tpfmap.h` include file.

Table 45. TPF/APPC Mapped Conversation Return Codes

Symbolic name	Meaning
CM_ALLOCATE_FAILURE_NO_RETRY	The conversation cannot be allocated on a session and the problem is not temporary. Do not retry the ALLOCATE.
CM_ALLOCATE_FAILURE_RETRY	The conversation cannot be allocated on a session but the problem may be temporary. Retry the ALLOCATE.
CM_CONVERSATION_TYPE_MISMATCH	This indicates that the remote LU rejected the allocation request because the remote transaction program does not support mapped conversations. The conversation is terminated.
CM_DEALLOCATED_ABEND	This can occur if the remote program calls <code>cmdeat</code> with <code>deallocate_type</code> set to <b>CM_DEALLOCATE_ABEND</b> . Or, if the remote program ends abnormally, the remote LU calls <code>cmdeat</code> with <code>deallocate_type</code> set to <b>CM_DEALLOCATE_ABEND</b> . The conversation is terminated. If the remote program was in <u>receive</u> state, some of the information sent by the local program may have been purged.
CM_DEALLOCATED_NORMAL	This occurs when the remote program calls <code>cmdeat</code> with: <ul style="list-style-type: none"><li><code>deallocate_type</code> set to <b>CM_DEALLOCATE_FLUSH</b></li><li><code>deallocate_type</code> set to <b>CM_DEALLOCATE_SYNC_LEVEL</b> and <code>sync_level</code> was <b>CM_NONE</b>.</li></ul> The conversation is terminated.
CM_OK	The function completed successfully.
CM_PARAMETER_ERROR	One of the parameters on the function or one of the associated characteristics contained invalid information. The source of the error is considered to be outside the control of the local program; for example, in the side information table. The conversation's state remains the same. Compare this with <b>CM_PROGRAM_PARAMETER_CHECK</b> .
CM_PRODUCT_SPECIFIC_ERROR	This indicates that an error occurred during TPF's internal processing of the function. The conversation is terminated.
CM_PROGRAM_ERROR_NO_TRUNC	This indicates that the remote program called <code>cmserr</code> while in <u>send</u> or <u>send-pending</u> (with <code>error_direction</code> set to <b>CM_SEND_ERROR</b> ) state. No truncation occurred. The local program remains in <u>receive</u> state.
CM_PROGRAM_ERROR_PURGING	This indicates that the remote program called <code>cmserr</code> while in <u>receive</u> , <u>send-pending</u> (with <code>error_direction</code> set to <b>CM_RECEIVE_ERROR</b> ), <u>confirm</u> , <u>confirm-send</u> , or <u>confirm-deallocate</u> state. If the remote program was in <u>receive</u> state but did not receive all the information sent by the local program, data may be purged at the local LU, remote LU, or both. The local program is placed in <u>receive</u> state.
CM_PROGRAM_PARAMETER_CHECK	One of the parameters on the function or one of the associated characteristics contained invalid information. The source of the error is considered to be under the control of the local program. No other returned variable contains valid information. The conversation's state is unchanged. Compare this with <b>CM_PARAMETER_ERROR</b> .
CM_PROGRAM_STATE_CHECK	This indicates that a local program called a function while in a state that does not allow that function. No other returned variables contain valid information. The conversation's state is unchanged.

Table 45. TPF/APPC Mapped Conversation Return Codes (continued)

Symbolic name	Meaning
CM_RESOURCE_FAILURE_NO_RETRY	The conversation was prematurely terminated because of some failure. The condition is not temporary; do not retry the transaction. The conversation is terminated.
CM_RESOURCE_FAILURE_RETRY	The conversation was prematurely terminated because of some failure. The condition may be temporary; retry the transaction. The conversation is terminated.
CM_SVC_ERROR_NO_TRUNC	This return code is possible only when the remote transaction program is using an LU 6.2 interface other than this one. The remote program issued a SEND_ERROR with type=SVC, the remote program was in send state, and the function did not truncate a logical record. This places the local program in <u>receive</u> state.
CM_SVC_ERROR_PURGING	This return code is possible only when the remote transaction program is using an LU 6.2 interface other than this one. The remote program issued a SEND_ERROR with type=SVC while it was in receive or confirm state and information was purged. The local program enters receive state.
CM_SVC_ERROR_TRUNC	This return code is possible only when the remote transaction program is using an LU 6.2 interface other than this one. The remote program issued a SEND_ERROR with type=SVC, the remote program was in send state, and the function truncated a logical record. This places the local program in <u>receive</u> state.
CM_SYNC_LVL_NOT_SUPPORTED_PGM	The remote LU rejected the allocation request because the local program specified a synchronization level the remote program does not support. The conversation is terminated.
CM_TPN_NOT_RECOGNIZED	The remote LU rejected the allocation request because it did not recognize the requested remote transaction program. The conversation is terminated.
CM_TP_NOT_AVAILABLE_NO_RETRY	The remote LU rejected the allocation request because it was unable to start the requested remote transaction program. The condition is not temporary; do not retry the allocation request. The conversation is terminated.
CM_TP_NOT_AVAILABLE_RETRY	The remote LU rejected the allocation request because it was unable to start the requested remote transaction program. The condition may be temporary; retry the allocation request. The conversation is terminated.
CM_UNSUCCESSFUL	The function issued by the local program did not execute successfully. The state of the conversation remains unchanged.

## cmaccp—Accept a Conversation

This function accepts an incoming conversation. It returns the conversation ID that identifies this conversation from now on.

### Format

```
#include <tpfmap.h>
void cmaccp (unsigned char *conversation_ID,
             signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array where the conversation ID is returned. You must specify this conversation ID on all subsequent functions for this conversation.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the cmaccp function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_STATE\_CHECK — No incoming conversation exists.

### Programming Considerations

- You can execute this function on any I-stream.
- The program must call the cmaccp function before any other functions for this conversation. In addition, the incoming conversation must be accepted before any others can be allocated.
- The value returned to **conversation\_ID** must be used on all other TPF/APPC mapped functions called for this conversation.
- There are no characteristics that modify this function.
- The following characteristics are initialized to match the values on the incoming conversation:
  - *conversation\_type*
  - *mode\_name*
  - *mode\_name\_length*
  - *partner\_LU\_name*
  - *partner\_LU\_name\_length*
  - *sync\_level*.
- The following characteristics are always initialized as follows:
  - *deallocate\_type* is set to **CM\_DEALLOCATE\_SYNC\_LEVEL**.
  - *error\_direction* is set to **CM\_RECEIVE\_ERROR**.
  - *prepare\_to\_receive\_type* is set to **CM\_PREP\_TO\_RECEIVE\_SYNC\_LEVEL**.
  - *send\_type* is set to **CM\_BUFFER\_DATA**.
- *TP\_name*, *TP\_name\_length*, and *return\_control* apply only to a `cmalloc` call.
- If the **return\_code** is CM\_OK, the conversation enters receive state.
- Each transaction program instance can issue only one cmaccp (ACCEPT\_CONVERSATION).

## Examples

The following example accepts in incoming conversation.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    rcode;
    .
    .
    .
cmapcp(convid,&rcode);
    .
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmapc—Allocate a Conversation” on page 800
- “cminit—Initialize a Conversation” on page 820.

## cmalloc–Allocate a Conversation

This function establishes a conversation with the partner program specified by the *TP\_name* and *partner\_LU\_name* characteristics. This function establishes only mapped conversations.

### Format

```
#include <tpfmap.h>
void      cmalloc(unsigned char *conversation_ID,
                  signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID assigned on the previous `cminit` (INITIALIZE) for this conversation.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmalloc` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_ALLOCATE\_FAILURE\_NO\_RETRY
- CM\_ALLOCATE\_FAILURE\_RETRY
- CM\_PARAMETER\_ERROR — The *mode\_name*, *TP\_name*, or *partner\_LU\_name* characteristic is invalid.
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The conversation is not in `initialize` state.
- CM\_UNSUCCESSFUL — No session is immediately available, and the *return\_control* characteristic is set to CM\_IMMEDIATE.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in `initialize` state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cminit` function.
- The first 4 bytes of the value supplied in **conversation\_ID** is stored in field EBCCBID in the ECB. This value changes with every ALLOCATE request.
- The following characteristics can affect this function:
  - *mode\_name*
  - *mode\_name\_length*
  - *partner\_LU\_name*
  - *partner\_LU\_name\_length*
  - *return\_control*
  - *sync\_level*
  - *TP\_name*
  - *TP\_name\_length*.

- If you are allocating a secondary LU (SLU) thread session, you must specify a single session mode name for the *partner\_LU\_name* characteristic. Parallel sessions are not supported for SLU threads.
- If the *return\_control* characteristic is set to **CM\_IMMEDIATE**, the `cma1lc` function does not wait if there is no contention winner session immediately available. If the characteristic is set to **CM\_WHEN\_SESSION\_ALLOCATED**, the ECB is suspended until a session becomes available.
- A session is activated for this conversation if all of the following conditions are true:
  - The *return\_control* characteristic is set to **CM\_WHEN\_SESSION\_ALLOCATED**
  - A session is not already available for the conversation
  - The session limits have *not* been reached for the remote LU and the mode name.

**Note:** PU 2.1 secondary LU (SLU) thread sessions cannot be activated from the TPF side. In this case, if the *partner\_LU\_name* characteristic specifies a local secondary LU (SLU) thread, that SLU must already be in session with the remote LU.

ALLOCATE must know what 2 LUs are involved to activate the session. One LU is specified with the *partner\_LU\_name* characteristic; the other LU is determined as follows:

- If you specify a remote LU that is in session with SLU threads, ALLOCATE selects the thread to be used for the session.
- If you specify a local LU that is a SLU thread, a remote LU must have been previously initialized for that SLU thread (with a CNOS INITIALIZE request).
- If you specify a parallel sessions mode name, the LU specified on the CNOS INITIALIZE request is used.
- The default TPF/APPC LU is used if the following conditions are true:
  - You specify single sessions
  - You are not using SLU threads
  - A CNOS INITIALIZE was not done.
- The LU specified on the CNOS INITIALIZE is used if the following conditions are true:
  - You specify single sessions
  - A CNOS INITIALIZE was done.
- The ALLOCATE request is queued until a session becomes available for use by this conversation if all of the following conditions are true:
  - The *return\_control* characteristic is set to **CM\_WHEN\_SESSION\_ALLOCATED**
  - A session is not already available for the conversation
  - The session limits have been reached for the remote LU and the mode name.
- If there is no session currently active and the *return\_control* characteristic is set to **CM\_WHEN\_SESSION\_ALLOCATED**, this verb uses the TPF system's EVENT and POST facility to suspend the ECB until a session is established.

If a session is not established in a certain amount of time, the program sends a failure return code to the transaction program. The amount of time that the system waits is determined by the value you specify for the TPALLOC parameter on the SNAKEY macro. See *TPF ACF/SNA Network Generation* for information about the SNAKEY macro.

## cmallc

- If the conversation is allocated to a contention loser session, this verb uses the TPF system's EVENT and POST facility to suspend the ECB until the program receives a BID response.  
If the program does not receive a BID response from the remote LU in a certain amount of time, an UNBIND is scheduled for this session, and the program sends a failure return code to the transaction program. The amount of time that the system waits is determined by the value you specify for the TPALLOC parameter on the SNAKEY macro. See *TPF ACF/SNA Network Generation* for information about the SNAKEY macro.
- If the ECB that issues cmallc does not already have a TCBID stored in field EBTCTCID in the ECB, the ALLOCATE request represents a new TPF transaction program instance, and a new TCBID is stored in EBTCTCID. If EBTCTCID does already have a value stored, the ALLOCATE request represents another conversation for the same transaction program instance, and EBTCTCID is not changed.
- The mode name SNASVCMG cannot be used by a user TPF transaction program.
- If the **return\_code** is CM\_OK, the conversation on the local transaction program side enters send state, and the conversation on the remote transaction program side enters receive state.
- If the local LU discovers an error with the allocation request, the program is notified immediately. However, if the remote LU rejects the allocation request, the program is notified by a later function's return code.
- The ATTACH is buffered until the buffer is full or a verb that implies the cmflus function is issued.

## Examples

The following example allocates a conversation.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    rcode;
    .
    .
    .
cmallc(convid,&rcode,);
    .
    .
    .
```

## Related Information

- "Return Codes for Mapped Conversation Functions" on page 796
- "cminit—Initialize a Conversation" on page 820
- "cmsmn—Set the Mode\_Name Characteristic" on page 842
- "cmspln—Set the Partner\_LU\_Name Characteristic" on page 844
- "cmsrc—Set the Return\_Control Characteristic" on page 848
- "cmssl—Set the Sync\_Level Characteristic" on page 850
- "cmstpn—Set the TP\_Name Characteristic" on page 854.



## cmcfm—Send a Confirmation Request

This function sends a confirmation request to the remote transaction program and waits for the confirmation reply. This allows the 2 programs to synchronize their processing.

### Format

```
#include <tpfmap.h>
void      cmcfm(unsigned char *conversation_ID,
               signed int *request_to_send_received,
               signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### request\_to\_send\_received

This is a pointer to a 4-byte field where the REQUEST\_TO\_SEND\_RECEIVED indication is placed. The REQUEST\_TO\_SEND\_RECEIVED indication is one of the following:

##### CM\_REQ\_TO\_SEND\_RECEIVED

This value indicates that a REQUEST\_TO\_SEND indication was received from the remote transaction program. This is a request that the local TPF transaction program enter receive state, so the remote program can enter send state.

##### CM\_REQ\_TO\_SEND\_NOT\_RECEIVED

This value indicates that a REQUEST\_TO\_SEND notification was not received.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmcfm` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK — The remote program responded with a `cmcfmd` (CONFIRMED).
- CM\_CONVERSATION\_TYPE\_MISMATCH
- CM\_DEALLOCATED\_ABEND
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_ERROR\_PURGING
- CM\_PROGRAM\_PARAMETER\_CHECK — The *sync\_level* is **CM\_NONE**, or the specified **conversation\_ID** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The conversation is not in send or send-pending state.
- CM\_RESOURCE\_FAILURE\_NO\_RETRY
- CM\_RESOURCE\_FAILURE\_RETRY
- CM\_SVC\_ERROR\_PURGING
- CM\_SYNC\_LVL\_NOT\_SUPPORTED\_PGM
- CM\_TP\_NOT\_AVAILABLE\_NO\_RETRY
- CM\_TP\_NOT\_AVAILABLE\_RETRY
- CM\_TPN\_NOT\_RECOGNIZED

## Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in send or send-pending state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cmunit` function.
- The *sync\_level* characteristic must be **CM\_CONFIRM**.
- If this function is called while the conversation is in send-pending state and **return\_code** is **CM\_OK**, the conversation enters send state.
- If the **return\_code** is **CM\_PROGRAM\_ERROR\_PURGING** or **CM\_SVC\_ERROR\_PURGING**, the conversation enters receive state.
- This call causes the local program's send buffer to be flushed.
- This function uses TPF's EVENT and POST facility to suspend the ECB until the program receives a confirmation reply. Because the ECB is suspended, release all unnecessary resources before issuing this function. Failure to do so can cause serious system performance.
- If the program does not receive a confirmation reply in a certain amount of time, TPF/APPC support issues a DEALLOCATE request to terminate the conversation. The amount of time that the system waits is determined by the value you specify for the TPRecv parameter on the SNAKEY macro. See *TPF ACF/SNA Network Generation* for information about the SNAKEY macro.
- When the **request\_to\_send\_received** parameter is **CM\_REQ\_TO\_SEND\_RECEIVED**, the remote program is requesting the local TPF transaction program to enter receive state and thereby place the remote program in send or send-pending state. The local TPF transaction program enters receive state by issuing `cmrcv` or `cmpttr`. The remote partner program enters the corresponding send or send-pending state when it calls `cmrcv` and receives the SEND indication on the **status\_received** parameter.

## Examples

The following example sends a confirmation request to the remote transaction program and waits for the confirmation reply.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    rtsr;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmcfm(convid,&rtsr,&rcode);
    .          /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmcfmd—Send a Confirmation Reply” on page 805
- “cmrts—Request Change to Send State” on page 829
- “cmssl—Set the Sync\_Level Characteristic” on page 850.

## cmcfmd—Send a Confirmation Reply

This function sends a confirmation reply to the remote transaction program. This allows the 2 programs to synchronize their processing. The local TPF transaction program can call this function when it receives a confirmation request from the remote transaction program.

### Format

```
#include <tpfmap.h>
void cmcfmd(unsigned char *conversation_ID,
             signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cmunit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmcfmd` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The conversation is not in confirm, confirm-send, or confirm-deallocate state.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in one of the following states:
  - confirm
  - confirm-send
  - confirm-deallocate.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cmunit` function.
- There are no characteristics that modify this function.
- When the **return\_code** is CM\_OK:
  - If the conversation was in confirm state, it changes to receive state.
  - If the conversation was in confirm-send state, it changes to send state.
  - If the conversation was in confirm-deallocate status, it terminates. The conversation is deallocated and no longer exists.
- A transaction program can call this function only as a reply to a confirmation request; the function cannot be called at any other time. Transaction programs can use this function for various application-level functions. For example, the remote program can send data followed by a confirmation request. When the local program receives the confirmation request, it can call this function as an indication that it received and processed the data without error.

## Examples

```
#include <tpfmap.h>
```

## Related Information

- 806 TPF V4R1 C/C++ Language Support User's Guide

## cmdeal–Deallocate a Conversation

This function deallocates the specified conversation from the transaction program. Depending on the value of the *deallocate\_type* and *sync\_level* characteristics, a FLUSH or CONFIRM may be done first. If the DEALLOCATE completes successfully, the conversation no longer exists and the **conversation\_ID** is no longer valid.

### Format

```
#include <tpfmap.h>
void cmdeal(unsigned char *conversation_ID,
            signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the *cminit* (INITIALIZE) or *cmaccp* (ACCEPT\_CONVERSATION) that started this conversation.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following are lists of return codes that can be returned to the program that called the *cmdeal* function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

When *deallocate\_type* is **CM\_DEALLOCATE\_SYNC\_LEVEL** and *sync\_level* is **CM\_NONE**, or *deallocate\_type* is **CM\_DEALLOCATE\_FLUSH** or **CM\_DEALLOCATE\_ABEND**, the possible return codes are:

- CM\_OK
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — *deallocate\_type* is **CM\_DEALLOCATE\_SYNC\_LEVEL** or **CM\_DEALLOCATE\_FLUSH** and the conversation is not in send or send-pending state.

When *deallocate\_type* is **CM\_DEALLOCATE\_SYNC\_LEVEL** and *sync\_level* is **CM\_CONFIRM**, or when *deallocate\_type* is **CM\_DEALLOCATE\_CONFIRM**, the possible return codes are:

- CM\_OK
- CM\_CONVERSATION\_TYPE\_MISMATCH
- CM\_DEALLOCATED\_ABEND
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_ERROR\_PURGING
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The program is not in send or send-pending state.
- CM\_RESOURCE\_FAILURE\_NO\_RETRY
- CM\_RESOURCE\_FAILURE\_RETRY
- CM\_SVC\_ERROR\_PURGING
- CM\_SYNC\_LVL\_NOT\_SUPPORTED\_PGM

## cmdeal

- CM\_TP\_NOT\_AVAILABLE\_NO\_RETRY
- CM\_TP\_NOT\_AVAILABLE\_RETRY
- CM\_TPN\_NOT\_RECOGNIZED.

## Programming Considerations

- You can execute this function on any I-stream.
- If *deallocate\_type* is **CM\_DEALLOCATE\_SYNC\_LEVEL**, **CM\_DEALLOCATE\_FLUSH**, or **CM\_DEALLOCATE\_CONFIRM**, the conversation must be in send or send-pending state.
- If *deallocate\_type* is **CM\_DEALLOCATE\_ABEND**, the conversation can be in any state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cmunit` function.
- The following characteristics can affect this function:
  - *sync\_level*
  - *deallocate\_type*.
- If **return\_code** is **CM\_OK**, the conversation terminates.
- If **return\_code** is **CM\_PROGRAM\_ERROR\_PURGING** or **CM\_SVC\_ERROR\_PURGING**, the conversation enters receive state.
- When the DEALLOCATE is complete, the conversation identified by the conversation ID is completed, and no further functions can be called for that conversation.
- If the transaction program exits without deallocating a conversation, the TPF system does not deallocate that conversation. The program should call this function for all conversations before exiting.
- This function flushes the send buffer.
- When *deallocate\_type* is **CM\_DEALLOCATE\_SYNC\_LEVEL** and *sync\_level* is **CM\_CONFIRM**, or if *deallocate\_type* is **CM\_DEALLOCATE\_CONFIRM**, and the return code indicates an error, the conversation is not deallocated.
- When *deallocate\_type* is **CM\_DEALLOCATE\_SYNC\_LEVEL** and *sync\_level* is **CM\_CONFIRM**, or if *deallocate\_type* is **CM\_DEALLOCATE\_CONFIRM**, TPF's EVENT and POST facility is used to suspend the ECB until the program receives a confirmation reply. Because the ECB is suspended, all unnecessary resources should be released before this verb is issued. Failure to do so can cause serious system degradation.
- If the *deallocate\_type* characteristic is set to **CM\_DEALLOCATE\_CONFIRM**, the normal response from the remote transaction program is **CONFIRMED**. However, if the remote program issues a **DEALLOCATE TYPE=ABEND**, the `cmdeal` **return\_code** is **CM\_DEALLOCATED\_ABEND**, and the conversation terminates. If the remote transaction program issues a **SEND\_ERROR**, the local program goes into receive state, and the conversation continues.
- If *deallocate\_type* is **CM\_DEALLOCATE\_CONFIRM**, and the program does not receive a confirmation reply in a certain amount of time, TPF/APPC support issues an unbind to terminate the conversation. The amount of time that the system waits is determined by the value you specify for the **TPRECV** parameter on the **SNAKEY** macro. See *TPF ACF/SNA Network Generation* for information about the **SNAKEY** macro.

## Examples

The following example deallocates a specified conversation.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmdeal(convid,&rcode);
    .          /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmsdt–Set the Deallocate\_Type Characteristic” on page 831.

## cmecs—Extract the Conversation State

This function returns the current value of the *conversation\_state* characteristic.

### Format

```
#include <tpfmap.h>
void cmecs(unsigned char *conversation_ID,
           signed int *conversation_state,
           signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### conversation\_state

This is a pointer to a 4-byte field where the current state of the conversation is to be placed. The possible values are:

- **CM\_INITIALIZE\_STATE**
- **CM\_SEND\_STATE**
- **CM\_RECEIVE\_STATE**
- **CM\_SEND\_PENDING\_STATE**
- **CM\_CONFIRM\_STATE**
- **CM\_CONFIRM\_SEND\_STATE**
- **CM\_CONFIRM\_DEALLOCATE\_STATE**

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmecs` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- **CM\_OK**
- **CM\_PROGRAM\_PARAMETER\_CHECK** — The specified **conversation\_ID** is invalid.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation can be in any state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cminit` function.
- This function does not modify *conversation\_state*.

### Examples

The following example gets the current conversation state.

```
#include <tpfmap.h>

unsigned char convid[8];
signed int state;
signed int rcode;
.
.
/* set conversation_ID with value returned from accept or initialize */
.
.
```



```
cmecs(convid,&state,&rcode);  
.  
.  
.  
/* normal processing path */
```

## Related Information

“Return Codes for Mapped Conversation Functions” on page 796.

## cmemn–Extract the Mode Name

This function returns the current value of the *mode\_name* and *mode\_name\_length* characteristics.

### Format

```
#include <tpfmap.h>
void      cmemn(unsigned char *conversation_ID,
                unsigned char *mode_name,
                signed int *mode_name_length,
                signed int *return_code);
```

#### **conversation\_ID**

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### **mode\_name**

This is a pointer to an array of characters where the mode name for this conversation is to be placed. The string must allow for as many as 8 characters. The previous contents of the string is overwritten on return.

#### **mode\_name\_length**

This is a pointer to a 4-byte field where the length of the mode name is to be placed. Trailing blanks are not included.

#### **return\_code**

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmemn` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- `CM_OK`
- `CM_PRODUCT_SPECIFIC_ERROR`
- `CM_PROGRAM_PARAMETER_CHECK` — The specified **conversation\_ID** is invalid.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation can be in any state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cminit` function.
- This function does not change the *mode\_name* or *mode\_name\_length* characteristics.
- If **return\_code** is not `CM_OK`, **mode\_name** and **mode\_name\_length** do not return meaningful values.

### Examples

The following example gets the mode name for the conversation.

```
#include <tpfmap.h>

    unsigned char convid[8];
    unsigned char name[8];
    signed int    length;
    signed int    rcode;
```

```

        .
        .
        /* set conversation_ID with value returned from accept or initialize */
        .
        .
        cmemn(convid,name,&length,&rcode);
        .                               /* normal processing path                               */
        .
        .

```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmsmn—Set the Mode\_Name Characteristic” on page 842.

## cmepln—Extract the Partner LU Name

This function returns the current value of the *partner\_LU\_name* and *partner\_LU\_name\_length* characteristics.

### Format

```
#include <tpfmap.h>
void cmepln(unsigned char *conversation_ID,
            unsigned char *partner_LU_name,
            signed int *partner_LU_name_length,
            signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the *cminit* (INITIALIZE) or *cmaccp* (ACCEPT\_CONVERSATION) that started this conversation.

#### partner\_LU\_name

This is a pointer to an array of characters where the fully qualified partner LU name for this conversation will be placed. The string must allow for as many as 17 characters. On return, the string contains the name of the LU's network, followed by a period, followed by the partner LU name. If there is no period, the entire string is the LU name. Trailing blanks are compressed. The previous contents of the string are overwritten.

#### partner\_LU\_name\_length

This is a pointer to a 4-byte field where the length of the mode name will be placed. Trailing blanks are not included.

#### return\_code

This is a pointer to a 4-byte field where the return code will be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the *cmepln* function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation can be in any state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the *cmaccp* or *cminit* function.
- This function does not change the *partner\_LU\_name* or *partner\_LU\_name\_length* characteristics.
- If **return\_code** is not CM\_OK, **partner\_LU\_name** returns no information.
- If **return\_code** is not CM\_OK, **partner\_LU\_name\_length** does not return a meaningful value.

### Examples

The following example gets the partner LU name for the conversation.

```

#include <tpfmap.h>

    unsigned char convid[8];
    unsigned char name[17];
    signed int    length;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmepIn(convid,name,&length,&rcode);
    .          /* normal processing path          */
    .
    .

```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmspln–Set the Partner\_LU\_Name Characteristic” on page 844.

## cmesl—Extract the Sync Level

This function returns the current value of the *sync\_level* characteristic.

### Format

```
#include <tpfmap.h>
void      cmesl(unsigned char *conversation_ID,
               signed int *sync_level,
               signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### sync\_level

This is a pointer to a 4-byte field where the current synchronization level of the conversation will be placed. The possible values are:

- **CM\_NONE**
- **CM\_CONFIRM.**

#### return\_code

This is a pointer to a 4-byte field where the return code will be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmesl` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- **CM\_OK**
- **CM\_PROGRAM\_PARAMETER\_CHECK** — The specified **conversation\_ID** is invalid.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation can be in any state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cminit` function.
- This function does not modify the *sync\_level* characteristic.
- If **return\_code** is not **CM\_OK**, **sync\_level** does not return a meaningful value.

### Examples

The following example gets the synchronization level for the conversation.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    sync;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmesl(convid,&sync,&rcode);
    .          /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmssl–Set the Sync\_Level Characteristic” on page 850.

## cmflus—Flush Data from Buffer of Local LU

This function allows the application to transmit any data from the local LU's data buffer, including an ATTACH from the `cmallc` function.

### Format

```
#include <tpfmap.h>
void      cmflus(unsigned char *conversation_ID,
                signed int *return_code);
```

#### **conversation\_ID**

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### **return\_code**

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmflus` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_CONVERSATION\_TYPE\_MISMATCH
- CM\_DEALLOCATED\_ABEND
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_ERROR\_PURGING
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The program is not in send or send-pending state.
- CM\_RESOURCE\_FAILURE\_NO\_RETRY
- CM\_RESOURCE\_FAILURE\_RETRY
- CM\_SVC\_ERROR\_PURGING
- CM\_SYNC\_LVL\_NOT\_SUPPORTED\_PGM
- CM\_TP\_NOT\_AVAILABLE\_NO\_RETRY
- CM\_TP\_NOT\_AVAILABLE\_RETRY
- CM\_TPN\_NOT\_RECOGNIZED.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in send or send-pending state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cminit` function.
- If this function is called while the conversation is in send-pending state and **return\_code** is CM\_OK, the conversation enters send state.
- This function is useful for optimization of processing between the local and remote transaction programs. The TPF/APPC support code buffers the information from consecutive `cmsend` functions until it has a sufficient amount of information for transmission. At that time, it transmits the buffered data. However, the local TPF transaction program can issue `cmflus` in order to cause the



The following example calls the `cmflus` function.

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmlloc—Allocate a Conversation” on page 800
- “cmsend—Send Data to Remote Transaction Program” on page 835
- “cmserr—Send Error Notification” on page 838
- “cmsst—Set the Send\_Type Characteristic” on page 852.

## cminit—Initialize a Conversation

This function allocates local resources for a conversation and initializes various characteristics before the `ALLOCATE` is done. The remote partner uses the `cmaccp` function (`ACCEPT_CONVERSATION`) to initialize the conversation on its end.

### Format

```
#include <tpfmap.h>
void      cminit (unsigned char *conversation_ID,
                  unsigned char *sym_dest_name,
                  signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array where the conversation ID is returned. You must specify this conversation ID on all subsequent functions for this conversation.

#### sym\_dest\_name

This is a pointer to an 8-character string, which specifies the symbolic destination name. This symbolic name is used to determine the remote transaction program name, partner LU name, and mode name for the session that is to carry the conversation. The side information table is searched for the symbolic name and the following characteristics are set from the entry:

- *partner\_LU\_name*
- *partner\_LU\_name\_length*
- *mode\_name*
- *mode\_name\_length*
- *TP\_name*
- *TP\_name\_length*.

If the symbolic name is less than 8 characters, it is left-justified and padded to the right with blanks. If the string contains all blanks, the side information table is not searched and the characteristics listed previously are not set. In this case, the program is expected to set the characteristics with the set functions before doing the `cmalloc` function.

**Note:** See the *TPF ACF/SNA Data Communications Reference* for more information about the side information table. See *TPF Operations* for more information about the `ZNSID` command.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cminit` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- `CM_OK`
- `CM_PRODUCT_SPECIFIC_ERROR`
- `CM_PROGRAM_PARAMETER_CHECK` — The specified **sym\_dest\_name** could not be found.

### Programming Considerations

- You can execute this function on any I-stream.
- This must be the first function called for any conversation initiated by the program.

- The value returned to **conversation\_ID** must be used on all other TPF/APPC mapped functions called for this conversation.
- There are no characteristics that modify this function.
- The following characteristics are initialized to match the values in the side information table:
  - *mode\_name*
  - *mode\_name\_length*
  - *partner\_LU\_name*
  - *partner\_LU\_name\_length*
  - *TP\_name*
  - *TP\_name\_length*.
- The following characteristics are always initialized as follows:
  - *conversation\_type* is set to **CM\_MAPPED\_CONVERSATION**.
  - *deallocate\_type* is set to **CM\_DEALLOCATE\_SYNC\_LEVEL**.
  - *error\_direction* is set to **CM\_RECEIVE\_ERROR**.
  - *prepare\_to\_receive\_type* is set to **CM\_PREP\_TO\_RECEIVE\_SYNC\_LEVEL**.
  - *return\_control* is set to **CM\_WHEN\_SESSION\_ALLOCATED**.
  - *send\_type* is set to **CM\_BUFFER\_DATA**.
  - *sync\_level* is set to **CM\_NONE**.
- If the **return\_code** is CM\_OK, the conversation enters initialize state.
- The `cm_init` function can be called more than once in order to have multiple conversations. The **conversation\_ID** returned is always different.
- You can use the SET functions to change the values of the characteristics before the ALLOCATE.

## Examples

The following example initializes a conversation.

```
#include <tpfmap.h>

    unsigned char convid[8];
    unsigned char name[8];
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cminit(convid,name,&rcode);
    .                               /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmaccp—Accept a Conversation” on page 798
- “cmallc—Allocate a Conversation” on page 800
- “cmsdt—Set the Deallocate\_Type Characteristic” on page 831
- “cmsmn—Set the Mode\_Name Characteristic” on page 842
- “cmspln—Set the Partner\_LU\_Name Characteristic” on page 844
- “cmsptr—Set the Prepare\_To\_Receive\_Type Characteristic” on page 846
- “cmsrc—Set the Return\_Control Characteristic” on page 848
- “cmssl—Set the Sync\_Level Characteristic” on page 850
- “cmsst—Set the Send\_Type Characteristic” on page 852

## **cminit**

- “cmstpncpy–Set the TP\_Name Characteristic” on page 854.

## cmptr—Prepare to Receive Data

This function changes the conversation from send or send-pending to receive state in preparation to receive data. The execution of this function includes the function of the FLUSH or CONFIRM verb.

### Format

```
#include <tpfmap.h>
void      cmptr(unsigned char *conversation_ID,
               signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID.

This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmptr` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

When *prepare\_to\_receive\_type* is set to **CM\_PREP\_TO\_RECEIVE\_FLUSH**, or *prepare\_to\_receive\_type* is set to **CM\_PREP\_TO\_RECEIVE\_SYNC\_LEVEL** and *sync\_level* is **CM\_NONE**, the possible return codes are:

- CM\_OK
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The conversation is not in send or send-pending state.

When *prepare\_to\_receive\_type* is set to **CM\_PREP\_TO\_RECEIVE\_CONFIRM**, or *prepare\_to\_receive\_type* is set to **CM\_PREP\_TO\_RECEIVE\_SYNC\_LEVEL** and *sync\_level* is **CM\_CONFIRM**, the possible return codes are:

- CM\_OK
- CM\_CONVERSATION\_TYPE\_MISMATCH
- CM\_DEALLOCATED\_ABEND
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_ERROR\_PURGING
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The conversation is not in send or send-pending state.
- CM\_RESOURCE\_FAILURE\_NO\_RETRY
- CM\_RESOURCE\_FAILURE\_RETRY
- CM\_SVC\_ERROR\_PURGING
- CM\_SYNC\_LVL\_NOT\_SUPPORTED\_PGM
- CM\_TP\_NOT\_AVAILABLE\_NO\_RETRY
- CM\_TP\_NOT\_AVAILABLE\_RETRY

## cmptr

- CM\_TPN\_NOT\_RECOGNIZED.

## Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in send or send-pending state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cmunit` function.
- The following characteristics can affect this function:
  - *prepare\_to\_receive\_type*
  - *sync\_level*
- If the **return\_code** is `CM_OK`, the conversation enters receive state.
- This call causes the local program's send buffer to be flushed.
- If *prepare\_to\_receive\_type* is **CM\_PREP\_TO\_RECEIVE\_CONFIRM**, the TPF system EVENT and POST facility is used to suspend the ECB until the program receives the confirmation reply. Because the ECB is suspended, all unnecessary resources should be released before this verb is issued. Failure to do so can cause serious system degradation.
- If *prepare\_to\_receive\_type* is **CM\_PREP\_TO\_RECEIVE\_CONFIRM** and the program does not receive a confirmation reply in a certain amount of time, TPF/APPC support issues a DEALLOCATE request to terminate the conversation. The amount of time that the system waits is determined by the value you specify for the `TPRECV` parameter on the `SNAKEY` macro. See *TPF ACF/SNA Network Generation* for information about the `SNAKEY` macro.
- The remote transaction program enters send, send-pending, or confirm-send state when it receives the SEND indication in the **status\_received** parameter of the `cmrcv` function. (See “`cmrcv`—Receive Information” on page 825.) The remote transaction program can then send data to the local TPF transaction program.

## Examples

The following example places the conversation in receive state.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmptr(convid,&rcode);
    .                               /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “`cmsptr`—Set the Prepare\_To\_Receive\_Type Characteristic” on page 846
- “`cmssl`—Set the Sync\_Level Characteristic” on page 850.

## cmrcv–Receive Information

This function waits for information to arrive on the specified conversation and then receives the information. If information is already available, the information is received without waiting. The information received may be data, conversation status, or a confirmation request.

### Format

```
#include <tpfmap.h>
void cmrcv(unsigned char *conversation_ID,
           unsigned char *buffer,
           signed int *requested_length,
           signed int *data_received,
           signed int *received_length,
           signed int *status_received,
           signed int *request_to_send_received,
           signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cmnit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### buffer

This is a pointer to an array of characters where the data is placed on return. The area only contains data if **return\_code** is `CM_OK` or `CM_DEALLOCATED_NORMAL` and **data\_received** is not **CM\_NO\_DATA\_RECEIVED**. The area must be large enough to hold the number of bytes specified by **requested\_length**.

#### requested\_length

This is a pointer to a 4-byte field that specifies the maximum amount of data the program is to receive. The value must be from 0–32 767.

#### data\_received

This is a pointer to a 4-byte field that indicates if the program received data. This indication is valid only if **return\_code** is `CM_OK` or `CM_DEALLOCATED_NORMAL`. Possible values are:

##### **CM\_NO\_DATA\_RECEIVED**

This value indicates that no data was received.

##### **CM\_COMPLETE\_DATA\_RECEIVED**

This value indicates that a complete logical record or the last remaining portion of a logical record was received.

##### **CM\_INCOMPLETE\_DATA\_RECEIVED**

This value indicates that less than a complete logical record was received by the local TPF transaction program. Another `cmrcv` must be called to get the remaining data.

#### received\_length

This is a pointer to a 4-byte field that, on return, contains the number of bytes of data that were received. If the program does not receive data, this parameter is invalid.

#### status\_received

This is a pointer to a 4-byte field that, on return, contains an indication of the conversation status. The possible values are:

**CM\_NO\_STATUS\_RECEIVED**

This value indicates that no conversation status was received by the program. Data may have been received.

**CM\_SEND\_RECEIVED**

This value indicates that the remote program has entered receive state. If **data\_received** is **CM\_COMPLETE\_DATA\_RECEIVED**, the local program enters send-pending state. If **data\_received** is **CM\_NO\_DATA\_RECEIVED**, the local program enters send state.

**CM\_CONFIRM\_RECEIVED**

This value indicates that the remote program requested the local program to do a cmcfmd function to synchronize processing. The local program must call a cmcfmd, cmserr, or cmdeat function (with *deallocate\_type* set to **CM\_DEALLOCATE\_ABEND**).

**CM\_CONFIRM\_SEND\_RECEIVED**

This value indicates that the remote program has entered receive state and requested confirmation. The local program must call a cmcfmd, cmserr, or cmdeat function (with *deallocate\_type* set to **CM\_DEALLOCATE\_ABEND**).

**CM\_CONFIRM\_DEALLOC\_RECEIVED**

This value indicates that the remote program has deallocated the conversation with confirmation requested. The local program must call a cmcfmd, cmserr, or cmdeat function (with *deallocate\_type* set to **CM\_DEALLOCATE\_ABEND**).

**request\_to\_send\_received**

This is a pointer to a 4-byte field where a REQUEST\_TO\_SEND notification is placed. Possible values are:

**CM\_REQ\_TO\_SEND\_RECEIVED**

This value indicates that the local program received a REQUEST\_TO\_SEND notification from the remote program. This is a request that the local program enter receive state, which allows the remote program to enter send state.

**CM\_REQ\_TO\_SEND\_NOT\_RECEIVED**

This value indicates that a REQUEST\_TO\_SEND notification was not received.

**return\_code**

This is a pointer to a 4-byte field where the return code is to be placed.

## Return Codes

The following is a list of return codes that can be returned to the program that called the cmrcv function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_CONVERSATION\_TYPE\_MISMATCH
- CM\_DEALLOCATED\_ABEND
- CM\_DEALLOCATED\_NORMAL
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_ERROR\_NO\_TRUNC
- CM\_PROGRAM\_ERROR\_PURGING
- CM\_PROGRAM\_PARAMETER\_CHECK — Either **conversation\_ID** or **requested\_length** is invalid.



- CM\_PROGRAM\_STATE\_CHECK — The conversation is not in send, send-pending, or receive state.
- CM\_RESOURCE\_FAILURE\_NO\_RETRY
- CM\_RESOURCE\_FAILURE\_RETRY
- CM\_SVC\_ERROR\_NO\_TRUNC
- CM\_SVC\_ERROR\_PURGING
- CM\_SYNC\_LVL\_NOT\_SUPPORTED\_PGM
- CM\_TP\_NOT\_AVAILABLE\_NO\_RETRY
- CM\_TP\_NOT\_AVAILABLE\_RETRY
- CM\_TPN\_NOT\_RECOGNIZED.

## Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in one of the following states:  
receive  
send  
send-pending.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the cmaccp or cminit function.
- There are no characteristics that modify this function.
- The conversation enters receive state if the conversation was in send or send-pending state, **return\_code** is CM\_OK, and **status\_received** is **CM\_NO\_STATUS\_RECEIVED**.
- The conversation enters send state if **status\_received** is **CM\_SEND\_RECEIVED** and **data\_received** is **CM\_NO\_DATA\_RECEIVED**.
- The conversation enters send-pending state if **status\_received** is **CM\_SEND\_RECEIVED** and **data\_received** is **CM\_COMPLETE\_DATA\_RECEIVED**.
- The conversation enters confirm state when **status\_received** is **CM\_CONFIRM\_RECEIVED**.
- The conversation enters confirm-send state when **status\_received** is **CM\_CONFIRM\_SEND\_RECEIVED**.
- The conversation enters confirm-deallocate state when **status\_received** is **CM\_CONFIRM\_DEALLOC\_RECEIVED**.
- The mapped conversation support must be able to write to the buffer.
- Both data and status can be returned on the same cmrcv. A REQUEST\_TO\_SEND also can be returned on the same call with data or status.
- If there is no data available, TPF's EVENT and POST facility is used to suspend the ECB until data arrives. Because the ECB is suspended, all unnecessary resources should be released before this verb is issued. Failure to do so can cause serious system degradation.
- If the program does not receive any data in a certain amount of time, the conversation is terminated and cmrcv returns with **return\_code** set to CM\_PRODUCT\_SPECIFIC\_ERROR. The amount of time that the system waits is determined by the value you specify for the TPrecv parameter of the SNAKEY macro. See *TPF ACF/SNA Network Generation* for information about the SNAKEY macro.
- If **return\_code** is CM\_PROGRAM\_STATE\_CHECK or CM\_PROGRAM\_PARAMETER\_CHECK when the function ends, no other variable contains valid information.

## cmrcv

- When the program receives a complete data record or the last portion of a data record, the **data\_received** parameter is set to **CM\_COMPLETE\_DATA\_RECEIVED**. The amount of data is less than or equal to the **requested\_length** parameter.
- When the program receives a partial data record, **data\_received** parameter is set to **CM\_INCOMPLETE\_DATA\_RECEIVED**. The amount of data is less than or equal to the **requested\_length** parameter.
- If **requested\_length** is set to 0, the function acts as usual, except no data is received once it is available.

## Examples

The following example waits for information to arrive on the specified conversation and then receives the information.

```
#include <tpfmap.h>

    unsigned char convid[8];
    unsigned char *buff;
    signed int    req_len;
    signed int    data_rec;
    signed int    rec_len;
    signed int    status;
    signed int    rtsr;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmrcv(convid,buff,&req_len,&data_rec,&rec_len,&status,&rtsr,&rcode);
    .                               /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmsend—Send Data to Remote Transaction Program” on page 835.

## cmrts–Request Change to Send State

This function notifies the remote transaction program that the local TPF transaction program is requesting to enter send state for the conversation. The conversation changes to send, send-pending, or confirm-send state when the local TPF program subsequently receives a SEND indication from the remote program.

### Format

```
#include <tpfmap.h>
void      cmrts(unsigned char *conversation_ID,
               signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmrts` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The conversation is not in send, receive, send-pending, confirm, confirm-send, or confirm-deallocate state.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in one of the following states:
  - receive
  - send
  - send-pending
  - confirm
  - confirm-send
  - confirm-deallocate.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cminit` function.
- After issuing the `cmrts` function, the local TPF transaction program must call the `cmrcv` function to wait for the SEND indication to arrive from the remote program. The SEND indication is received in the **status\_received** parameter of the `cmrcv` function. (See “`cmrcv`–Receive Information” on page 825.)
 

For this program to receive permission to send data, the remote program must do one of the following:

  - Issue a RECEIVE verb
  - Issue a PREPARE\_TO\_RECEIVE verb
- The remote program receives the REQUEST\_TO\_SEND notification when it does a function with a **request\_to\_send\_received** parameter.

## cmrts

- If there is already a REQUEST\_TO\_SEND notification waiting to be given to the remote program, any additional notifications arriving on the same conversation are discarded.

## Examples

The following example notifies the remote transaction program that the local transaction program is requesting to enter send state for the conversation.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmrts(convid,&rcode);                /* normal processing path      */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmrcv—Receive Information” on page 825.

## cmsdt–Set the Deallocate\_Type Characteristic

This function sets the *deallocate\_type* characteristic. It overrides the default value set by the *cmunit* or *cmaccp* function.

### Format

```
#include <tpfmap.h>
void cmsdt(unsigned char *conversation_ID,
           signed int *deallocate_type,
           signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the *cmunit* (INITIALIZE) or *cmaccp* (ACCEPT\_CONVERSATION) that started this conversation.

#### deallocate\_type

This is a pointer to a 4-byte field that specifies the type of deallocation to be done when the program issues a *cmdeat* function. Use one of the following values:

##### CM\_DEALLOCATE\_SYNC\_LEVEL

This value specifies that the *sync\_level* characteristic should be used to determine the type of deallocate. If *sync\_level* is **CM\_NONE**, the function executes a *cmflus* and then deallocates the conversation normally. If *sync\_level* is **CM\_CONFIRM**, the function executes a *cmcfm*. If the CONFIRM is successful, the conversation is deallocated normally. If the CONFIRM fails, the return code determines the state of the conversation.

##### CM\_DEALLOCATE\_FLUSH

This value specifies to do a *cmflus* function and then deallocate the conversation normally.

##### CM\_DEALLOCATE\_CONFIRM

This value specifies to do a *cmcfm* function. If the CONFIRM is successful, the conversation is deallocated normally. If the CONFIRM fails, the return code determines the state of the conversation.

##### CM\_DEALLOCATE\_ABEND

This value specifies that the conversation should be unconditionally and abnormally deallocated. If the conversation is in *send* state, the function executes a *cmflus*. If the conversation is in *receive* state, data may be purged.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the *cmsdt* function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — This indicates one of the following:
  - The specified **conversation\_ID** is invalid.
  - The specified **deallocate\_type** is invalid.
  - The *sync\_level* characteristic is **CM\_NONE**, and **deallocate\_type** is **CM\_DEALLOCATE\_CONFIRM**.

## Programming Considerations

- You can execute this function on any I-stream.
- The conversation can be in any state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cminit` function.
- This function changes the value of the *deallocate\_type* characteristic. The *sync\_level* characteristic can also determine the type of deallocation to be performed.
- Use **CM\_DEALLOCATE\_SYNC\_LEVEL** when the desired type of deallocate should change depending on the *sync\_level* characteristic.
- Use **CM\_DEALLOCATE\_ABEND** when a conversation should be unconditionally deallocated regardless of its synchronization level and state.
- This function causes no state change.
- If **return\_code** is not `CM_OK`, then the *deallocate\_type* characteristic remains unchanged.

## Examples

The following example changes the *deallocate\_type* characteristic.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    type;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmsdt(convid,&type,&rcode);
    .                /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmaccp—Accept a Conversation” on page 798
- “cminit—Initialize a Conversation” on page 820
- “cmdeal—Deallocate a Conversation” on page 807
- “cmssl—Set the Sync\_Level Characteristic” on page 850.

## cmsed—Set the `error_direction` Characteristic

This function sets the *error\_direction* characteristic. It overrides the default value set by the `cminit` or `cmaccp` function.

### Format

```
#include <tpfmap.h>
void cmsed(unsigned char *conversation_ID,
           signed int *error_direction,
           signed int *return_code);
```

#### **conversation\_ID**

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### **error\_direction**

This is a pointer to a 4-byte field that specifies the type of notification the remote program receives after the local program calls `cmserr` in send-pending state. Use one of the following values:

##### **CM\_RECEIVE\_ERROR**

This value specifies that, if the local program calls `cmserr` in send-pending state, the remote program is notified of the error by the `CM_PROGRAM_ERROR_PURGING` return code.

##### **CM\_SEND\_ERROR**

This value specifies that, if the local program calls `cmserr` in send-pending state, the remote program is notified of the error by the `CM_PROGRAM_ERROR_NO_TRUNC` return code.

#### **return\_code**

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmsed` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- `CM_OK`
- `CM_PRODUCT_SPECIFIC_ERROR`
- `CM_PROGRAM_PARAMETER_CHECK` — This indicates one of the following:
  - The specified **conversation\_ID** is invalid.
  - The specified **error\_direction** is invalid.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation can be in any state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cminit` function.
- This function changes the value of the *error\_direction* characteristic.
- The *error\_direction* characteristic takes effect only when a conversation is in send-pending state and affects only the `cmserr` (SEND\_ERROR) function. It resolves an ambiguous error condition that can occur when a program receives both a change of direction (permission to send) and data on a `cmrcv` call.

## cmsed

When `cmrcv` returns **data\_received** set to **CM\_COMPLETE\_DATA\_RECEIVED** together with **status\_received** set to **CM\_SEND\_RECEIVED**, the conversation is placed in send-pending state. Two possible error conditions can occur:

- The local program finds an error with the data it just received.
- The local program finds an error while processing the data to be sent to the remote program.

Use the *error\_direction* characteristic to indicate which of these 2 conditions occurred. If the error is in the received data, set the *error\_direction* to **CM\_RECEIVE\_ERROR** and the remote program receives a **return\_code** of **CM\_PROGRAM\_ERROR\_PURGING**. If the error is in the data to be sent, set the *error\_direction* to **CM\_SEND\_ERROR**, and the remote program receives a **return\_code** of **CM\_PROGRAM\_ERROR\_NO\_TRUNC**.

- This function causes no state change.
- If **return\_code** is not **CM\_OK**, the *error\_direction* characteristic remains unchanged.

## Examples

The following example changes the *error\_direction* characteristic.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    err_dir;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmsed(convid,&err_dir,&rcode);
    .                               /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmaccp—Accept a Conversation” on page 798
- “cminit—Initialize a Conversation” on page 820
- “cmserr—Send Error Notification” on page 838.



## cmsend—Send Data to Remote Transaction Program

This function sends data to the remote transaction program.

### Format

```
#include <tpfmap.h>
void      cmsend(unsigned char *conversation_ID,
                unsigned char *buffer,
                signed int *send_length,
                signed int *request_to_send_received,
                signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### buffer

This is a pointer to a location in memory that contains the data to be sent. The length of the data is specified by the **send\_length** parameter.

#### send\_length

This is a pointer to a 4-byte field that contains the number of bytes of data that are to be sent. This value can be from 0–32 767. If this parameter is 0, a null data record is sent.

#### request\_to\_send\_received

This is a pointer to a 4-byte field where a REQUEST\_TO\_SEND notification is placed. Possible values are:

##### CM\_REQ\_TO\_SEND\_RECEIVED

This value indicates that the local program received a REQUEST\_TO\_SEND notification from the remote program. This is a request that the local program enter receive state, which allows the remote program to enter send state.

##### CM\_REQ\_TO\_SEND\_NOT\_RECEIVED

This value indicates that a REQUEST\_TO\_SEND notification was not received.

#### return\_code

This is a pointer to a 4-byte field where the return code will be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmsend` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_CONVERSATION\_TYPE\_MISMATCH
- CM\_DEALLOCATED\_ABEND
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_ERROR\_PURGING
- CM\_PROGRAM\_PARAMETER\_CHECK — Either **conversation\_ID** or **send\_length** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The program is not in send or send-pending state.
- CM\_RESOURCE\_FAILURE\_NO\_RETRY

## cmsend

- CM\_RESOURCE\_FAILURE\_RETRY
- CM\_SVC\_ERROR\_PURGING
- CM\_SYNC\_LVL\_NOT\_SUPPORTED\_PGM
- CM\_TP\_NOT\_AVAILABLE\_NO\_RETRY
- CM\_TP\_NOT\_AVAILABLE\_RETRY
- CM\_TPN\_NOT\_RECOGNIZED.

## Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in send or send-pending state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cmunit` function.
- The following characteristics can affect this function:
  - *send\_type*
  - *prepare\_to\_receive\_type*
  - *deallocate\_type*.
  - *sync\_level*
- The conversation enters receive state when `cmsend` is called with *send\_type* set to **CM\_SEND\_AND\_PREP\_TO\_RECEIVE** and **return\_code** is `CM_OK`.
- The conversation terminates when `cmsend` is called with *send\_type* set to **CM\_SEND\_AND\_DEALLOCATE** and **return\_code** is `CM_OK`.
- The conversation enters send state if the conversation was in send-pending state, **return\_code** is `CM_OK`, and the *send\_type* characteristic is set to **CM\_BUFFER\_DATA**, **CM\_SEND\_AND\_FLUSH**, or **CM\_SEND\_AND\_CONFIRM**.
- When the **request\_to\_send\_received** parameter is **CM\_REQ\_TO\_SEND\_RECEIVED**, the remote program is requesting the local TPF transaction program to enter receive state and thereby place the remote program in send or send-pending state. The local TPF transaction program enters receive state by issuing `cmrcv` or `cmptr`. The remote partner program enters the corresponding send or send-pending state when it calls `cmrcv` and receives the **SEND** indication on the **status\_received** parameter.
- The data pointed to by **buffer** is plain data with no header or length fields. The length is determined from the **send\_length** parameter. It is treated as a single, complete logical record.
- The entire storage area referred to by the **buffer** and **send\_length** parameters must be addressable by the TPF/APPC support code.

## Examples

The following example sends data to the remote transaction program.

```
#include <tpfmap.h>

    unsigned char convid[8];
    unsigned char *buff;
    signed int    len;
    signed int    rtsr;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
```

```
cmsend(convid,buff,&len,&rtsr,&rcode);  
.  
.  
.  
/* normal processing path */
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmsdt–Set the Deallocate\_Type Characteristic” on page 831
- “cmsptr–Set the Prepare\_To\_Receive\_Type Characteristic” on page 846
- “cmsst–Set the Send\_Type Characteristic” on page 852.

## cmserr–Send Error Notification

This function informs the remote transaction program that the local TPF transaction program detected an error.

### Format

```
#include <tpfmap.h>
void cmserr(unsigned char *conversation_ID,
            signed int *request_to_send_received,
            signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cmnit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### request\_to\_send\_received

This is a pointer to a 4-byte field where a REQUEST\_TO\_SEND notification is placed. Possible values are:

##### CM\_REQ\_TO\_SEND\_RECEIVED

This value indicates that the local program received a REQUEST\_TO\_SEND notification from the remote program. This is a request that the local program enter receive state, which allows the remote program to enter send state.

##### CM\_REQ\_TO\_SEND\_NOT\_RECEIVED

This value indicates that a REQUEST\_TO\_SEND notification was not received.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmserr` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

When the conversation is in send state, the possible return codes are:

- CM\_OK
- CM\_CONVERSATION\_TYPE\_MISMATCH
- CM\_DEALLOCATED\_ABEND
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_ERROR\_PURGING
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_RESOURCE\_FAILURE\_NO\_RETRY
- CM\_RESOURCE\_FAILURE\_RETRY
- CM\_SVC\_ERROR\_PURGING
- CM\_SYNC\_LVL\_NOT\_SUPPORTED\_PGM
- CM\_TP\_NOT\_AVAILABLE\_NO\_RETRY
- CM\_TP\_NOT\_AVAILABLE\_RETRY
- CM\_TPN\_NOT\_RECOGNIZED.

When the conversation is in receive state, the possible return codes are:

- CM\_OK
- CM\_DEALLOCATED\_NORMAL
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_RESOURCE\_FAILURE\_NO\_RETRY
- CM\_RESOURCE\_FAILURE\_RETRY.

When the conversation is in send-pending state, the possible return codes are:

- CM\_OK
- CM\_DEALLOCATED\_ABEND
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_ERROR\_PURGING
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_RESOURCE\_FAILURE\_NO\_RETRY
- CM\_RESOURCE\_FAILURE\_RETRY
- CM\_SVC\_ERROR\_PURGING.

When the conversation is in confirm, confirm-send, or confirm-deallocate state, possible return codes are:

- CM\_OK
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_RESOURCE\_FAILURE\_NO\_RETRY
- CM\_RESOURCE\_FAILURE\_RETRY.

If the conversation is not in one of the states listed previously, the only possible return codes are:

- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_PROGRAM\_STATE\_CHECK.

## Programming Considerations

- You can execute this function on any I-stream.
- The conversation can be in any state except initialize.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cmunit` function.
- When the conversation is in send-pending state, the *error\_direction* characteristic affects the way an error is reported to the remote program.

If *error\_direction* is set to **CM\_RECEIVE\_ERROR**, the **return\_code** of the subsequent mapped conversation verb received by the remote program is set to **CM\_PROGRAM\_ERROR\_PURGING**.

If *error\_direction* is set to **CM\_SEND\_ERROR**, the **return\_code** of the subsequent mapped conversation verb received by the remote program is set to **CM\_PROGRAM\_ERROR\_NO\_TRUNC**. (See “`cmsed`—Set the Error\_Direction Characteristic” on page 833 for more information on the send-pending state and the *error\_direction* characteristic.)

## cmserr

- When the conversation is in send or receive state, the remote program is informed of the error by one of the following return codes:
  - **CM\_PROGRAM\_ERROR\_NO\_TRUNC** — This indicates that the local program is in send state and found an error while processing data to be sent to the remote program.
  - **CM\_PROGRAM\_ERROR\_PURGING** — This indicates that the local program is in receive state and found an error in the data received from the remote program.
- When **return\_code** is **CM\_OK**, the local program enters send state.
- If the conversation was in send state, this call causes the local program's send buffer to be flushed.
- The transaction programs can use this function for various application level functions. For example, the program can call this function to truncate an incomplete logical record, to inform the remote program of an error it detected in data it received, or to reject a confirmation request.
- If the local TPF transaction program calls **cmserr** while in receive state, purging of all buffered input occurs. The incoming information that is purged includes both data (logical records) and confirmation requests. In addition, if the confirmation request was due to the remote transaction program's calling **cmdeal** with *deallocate\_type* set to **CM\_DEALLOCATE\_CONFIRM**, the **DEALLOCATE** request is also purged. After issuing the **cmserr** function from receive state, the local transaction program is in send state, and the remote transaction program is in receive state.
- When the **request\_to\_send\_received** parameter is **CM\_REQ\_TO\_SEND\_RECEIVED**, the remote program is requesting the local TPF transaction program to enter receive state and thereby place the remote program in send or send-pending state. The local TPF transaction program enters receive state by issuing **cmrcv** or **cmptr**. The remote partner program enters the corresponding send or send-pending state when it calls **cmrcv** and receives the **SEND** indication on the **status\_received** parameter.
- This verb resets or cancels posting. If posting is active and the conversation was posted, or if posting is active and the conversation was not posted, posting is canceled.
- The truncation implied by the use of this verb can cause any of the following to be truncated:
  - Data sent by means of the **cmsend** function
  - Confirmation requests sent by means of **cmcfm**, **cmptr**, or **cmdeal**
  - Deallocation request sent by means of **cmdeal** with *deallocate\_type* set to **CM\_DEALLOCATE\_CONFIRM**.

## Examples

The following example informs the remote transaction program that the local TPF transaction program detected an error.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    rtsr;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
```

```
cmserr(convid,&rtsr,&rcode);  
.  
.  
.  
/* normal processing path */
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmsed—Set the Error\_Direction Characteristic” on page 833
- “cmsend—Send Data to Remote Transaction Program” on page 835
- “cmrts—Request Change to Send State” on page 829.

## cmsmn—Set the Mode\_Name Characteristic

This function sets the *mode\_name* and *mode\_name\_length* characteristics. It overrides the default value set by the `cminit` function.

### Format

```
#include <tpfmap.h>
void cmsmn(unsigned char *conversation_ID,
           unsigned char *mode_name,
           signed int *mode_name_length,
           signed int *return_code);
```

#### **conversation\_ID**

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) that started this conversation.

#### **mode\_name**

This is a pointer to an array of characters that specifies the new mode name. This mode name determines the network properties for the session to be allocated for the conversation.

#### **mode\_name\_length**

This is a pointer to a 4-byte field that contains the length of the new mode name. The value can be from 0–8 bytes. If the value is 0, the mode name is set to NULL, and the name specified by **mode\_name** is ignored.

#### **return\_code**

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmsmn` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — Either **conversation\_ID** or **mode\_name\_length** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The program is not in initialize state.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in initialize state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cminit` function.
- This function changes the value of the *mode\_name* and *mode\_name\_length* characteristics.
- This function causes no state change.
- If the new mode name is invalid, it is not detected until the `ALLOCATE` is attempted.
- If **return\_code** is not CM\_OK, the *mode\_name* and *mode\_name\_length* characteristics remain unchanged.



## Examples

The following example changes the *mode\_name* and *mode\_name\_length* characteristics.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    len;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from initialize */
    .
    len = 5;
    .
cmsmn(convid,"MODE1",&len,&rcode);
    .                               /* normal processing path      */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmallc—Allocate a Conversation” on page 800
- “cminit—Initialize a Conversation” on page 820.

## cmspln–Set the Partner\_LU\_Name Characteristic

This function sets the *partner\_LU\_name* and *partner\_LU\_name\_length* characteristics. It overrides the default value set by the `cminit` function.

### Format

```
#include <tpfmap.h>
void      cmspln(unsigned char *conversation_ID,
               unsigned char *partner_LU_name,
               signed int *partner_LU_name_length,
               signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) that started this conversation.

#### partner\_LU\_name

This is a pointer to an array of characters that specifies the new partner LU name. This partner LU name specifies the remote LU or local secondary LU (SLU) thread to be contacted when the ALLOCATE is executed. The string can be as many as 17 characters, consisting of the name of the LU's network, followed by a period, followed by the partner LU name. The partner LU name can be as many as 8 characters. If there is no period, the TPF system assumes only an LU name was supplied.

#### partner\_LU\_name\_length

This is a pointer to a 4-byte field that contains the length of the new partner LU name. The value can be from 1 to 17 bytes.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmspln` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — Either **conversation\_ID** or **partner\_LU\_name\_length** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The program is not in initialize state.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in initialize state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cminit` function.
- This function changes the value of the *partner\_LU\_name* and *partner\_LU\_name\_length* characteristics.
- This function causes no state change.
- If the new LU name is invalid, it is not detected until the ALLOCATE is attempted.
- If **return\_code** is not CM\_OK, the *partner\_LU\_name* and *partner\_LU\_name\_length* characteristics remain unchanged.

- See the programming considerations for `cmalloc` (“`cmalloc—Allocate a Conversation`” on page 800) for additional considerations when setting this characteristic.

## Examples

The following example changes the *partner\_LU\_name* and *partner\_LU\_name\_length* characteristics.

```
#include <tpfmap.h>

    unsigned char convid[8];
    unsigned char name[17];
    signed int    len;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from initialize */
    .
    .
cmspln(convid,name,&len,&rcode);
    .                               /* normal processing path      */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “`cmalloc—Allocate a Conversation`” on page 800
- “`cminit—Initialize a Conversation`” on page 820.

## cmsptr–Set the Prepare\_To\_Receive\_Type Characteristic

This function sets the *prepare\_to\_receive\_type* characteristic. It overrides the default value set by the `cminit` or `cmaccp` function.

### Format

```
#include <tpfmap.h>
void cmsptr(unsigned char *conversation_ID,
            signed int *prepare_to_receive_type,
            signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### prepare\_to\_receive\_type

This is a pointer to a 4-byte field that specifies how and when the conversation should enter *receive* state when the program issues a `cmsptr` function. Use one of the following values:

##### CM\_PREP\_TO\_RECEIVE\_SYNC\_LEVEL

This value specifies that the *sync\_level* characteristic should be used to determine the type of *prepare\_to\_receive* to be done. If *sync\_level* is **CM\_NONE**, the function executes a `cmflus` and then enters *receive* state. If *sync\_level* is **CM\_CONFIRM**, the function executes a `cmcfm`. If the CONFIRM is successful, the conversation enters *receive* state. If the CONFIRM fails, the return code determines the state of the conversation.

##### CM\_PREP\_TO\_RECEIVE\_FLUSH

This value specifies to do a `cmflus` function and then enter *receive* state.

##### CM\_PREP\_TO\_RECEIVE\_CONFIRM

This value specifies to do a `cmcfm` function. If the CONFIRM is successful, the conversation enters *receive* state. If the CONFIRM fails, the return code determines the state of the conversation.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmsptr` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- **CM\_OK**
- **CM\_PROGRAM\_PARAMETER\_CHECK** — This indicates one of the following:
  - The specified **conversation\_ID** is invalid.
  - The specified **prepare\_to\_receive\_type** is invalid.
  - The *sync\_level* characteristic is **CM\_NONE**, and **prepare\_to\_receive\_type** is **CM\_PREP\_TO\_RECEIVE\_CONFIRM**.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation can be in any state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cminit` function.

- This function changes the value of the *prepare\_to\_receive\_type* characteristic. The *sync\_level* characteristic can also determine the type of *prepare\_to\_receive* to be performed.
- This function causes no state change.
- If **return\_code** is not CM\_OK, the *prepare\_to\_receive\_type* characteristic remains unchanged.

## Examples

The following example changes the *prepare\_to\_receive\_type* characteristic.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    type;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmsptr(convid,&type,&rcode);
    .          /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmaccp—Accept a Conversation” on page 798
- “cminit—Initialize a Conversation” on page 820
- “cmptr—Prepare to Receive Data” on page 823.

## cmsrc—Set the Return\_Control Characteristic

This function sets the *return\_control* characteristic. It overrides the default value set by the `cminit` function.

### Format

```
#include <tpfmap.h>
void cmsrc(unsigned char *conversation_ID,
           signed int *return_control,
           signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) that started this conversation.

#### return\_control

This is a pointer to a 4-byte field that specifies when an ALLOCATE is to return control to the user. Use one of the following values:

##### CM\_WHEN\_SESSION\_ALLOCATED

This value specifies that control should not be returned until a session is found for the conversation.

##### CM\_IMMEDIATE

This value specifies that a session is to be allocated only if one is immediately available. If a session is not available, the ALLOCATE terminates with a return code of CM\_UNSUCCESSFUL.

#### return\_code

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmsrc` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_PROGRAM\_PARAMETER\_CHECK — Either **conversation\_ID** or **return\_control** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The program is not in initialize state.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in initialize state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cminit` function.
- This function changes the value of the *return\_control* characteristic.
- This function causes no state change.
- If **return\_code** is not CM\_OK, the *return\_control* characteristic remains unchanged.

### Examples

The following example changes the *return\_control* characteristic.

```

#include <tpfmap.h>

    unsigned char convid[8];
    signed int    control;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from initialize */
    .
    .
cmsrc(convid,&control,&rcode);
    .                               /* normal processing path      */
    .
    .

```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmallc—Allocate a Conversation” on page 800
- “cminit—Initialize a Conversation” on page 820.

## cmssl–Set the **Sync\_Level** Characteristic

This function sets the *sync\_level* characteristic. It overrides the default value set by the `cm_init` function.

### Format

```
#include <tpfmap.h>
void      cmssl(unsigned char *conversation_ID,
               signed int *sync_level,
               signed int *return_code);
```

#### **conversation\_ID**

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cm_init` (INITIALIZE) that started this conversation.

#### **sync\_level**

This is a pointer to a 4-byte field that specifies the synchronization level the local and remote programs can use for this conversation. Use one of the following values:

##### **CM\_NONE**

This value specifies that no synchronization processing is to be done. Parameters, characteristics, and functions related to confirmation are not recognized.

##### **CM\_CONFIRM**

This value specifies that programs can perform confirmation processing for this conversation.

#### **return\_code**

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmssl` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- **CM\_OK**
- **CM\_PROGRAM\_PARAMETER\_CHECK** — This indicates one of the following:
  - The specified **conversation\_ID** is invalid.
  - The specified **sync\_level** is invalid.
  - The *sync\_level* characteristic is **CM\_NONE**, and *send\_type*, *prepare\_to\_receive\_type*, or *deallocate\_type* is set to use confirmation.
- **CM\_PROGRAM\_STATE\_CHECK** — The program is not in initialize state.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in initialize state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cm_init` function.
- This function changes the value of the *sync\_level* characteristic.
- This function causes no state change.
- If **return\_code** is not **CM\_OK**, the *sync\_level* characteristic remains unchanged.



## Examples

The following example changes the *sync\_level* characteristic.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    sync;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from initialize */
    .
    .
cmssl(convid,&sync,&rcode);
    .          /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmallc—Allocate a Conversation” on page 800
- “cminit—Initialize a Conversation” on page 820
- “cmcfm—Send a Confirmation Request” on page 803
- “cmcfmd—Send a Confirmation Reply” on page 805.

## cmsst–Set the Send\_Type Characteristic

This function sets the *send\_type* characteristic. It overrides the default value set by the `cminit` or `cmaccp` function.

### Format

```
#include <tpfmap.h>
void cmsst(unsigned char *conversation_ID,
           signed int *send_type,
           signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### send\_type

This is a pointer to a 4-byte field that, when a program issues a `cmsend` function, specifies the information that will be sent to the remote program in addition to the data supplied to the `cmsend` function. This parameter also specifies whether the data will be buffered or sent immediately. Use one of the following values:

##### CM\_BUFFER\_DATA

This value specifies that the supplied data cannot be sent until a sufficient quantity is accumulated.

##### CM\_SEND\_AND\_FLUSH

This value specifies that no additional information is to be sent to the remote program. The function calls a `cmsend` followed by a `cmflush`.

##### CM\_SEND\_AND\_CONFIRM

This value specifies that the data is to be sent immediately, followed by a confirmation request.

##### CM\_SEND\_AND\_PREP\_TO\_RECEIVE

This value specifies that the data is to be sent immediately and the local transaction program is to be placed in receive state. After receiving all the data, the remote transaction program enters send-pending or confirm-send state.

##### CM\_SEND\_AND\_DEALLOCATE

This value specifies that the data is to be sent immediately. Then the conversation is deallocated.

#### return\_code

This is a pointer to a 4-byte field where the return code will be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmsst` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- `CM_OK`
- `CM_PRODUCT_SPECIFIC_ERROR`
- `CM_PROGRAM_PARAMETER_CHECK` — This indicates one of the following:
  - The specified **conversation\_ID** is invalid.
  - The specified **send\_type** is invalid.
  - The *sync\_level* characteristic is **CM\_NONE**, and **send\_type** is set to **CM\_SEND\_AND\_CONFIRM**.

## Programming Considerations

- You can execute this function on any I-stream.
- The conversation can be in any state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cmunit` function.
- This function changes the value of the *send\_type* characteristic.
- This function causes no state change.
- If **return\_code** is not `CM_OK`, the *send\_type* characteristic remains unchanged.
- All the values of **send\_type** that indicate dual functions are to be performed are equivalent to executing the 2 functions sequentially. For example, the value **CM\_SEND\_AND\_FLUSH** causes `cmsend` to act like a `cmsend` function followed by a `cmflus` function.

## Examples

The following example changes the *send\_type* characteristic.

```
#include <tpfmap.h>

    unsigned char convid[8];
    signed int    type;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmsst(convid,&type,&rcode);
    .          /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmaccp—Accept a Conversation” on page 798
- “cmunit—Initialize a Conversation” on page 820
- “cmcfm—Send a Confirmation Request” on page 803
- “cmdeal—Deallocate a Conversation” on page 807
- “cmflus—Flush Data from Buffer of Local LU” on page 818
- “cmptr—Prepare to Receive Data” on page 823
- “cmsend—Send Data to Remote Transaction Program” on page 835.

## cmstpn—Set the TP\_Name Characteristic

This function sets the *TP\_name* and *TP\_name\_length* characteristics. It overrides the default value set by the `cminit` function.

### Format

```
#include <tpfmap.h>
void      cmstpn(unsigned char *conversation_ID,
                 unsigned char *TP_name,
                 signed int *TP_name_length,
                 signed int *return_code);
```

#### **conversation\_ID**

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) that started this conversation.

#### **TP\_name**

This is a pointer to a string of as many as 64 characters. This string specifies the remote transaction program to be contacted when the `ALLOCATE` is executed.

#### **TP\_name\_length**

This is a pointer to a 4-byte field that contains the length of the new transaction program name. The value can be from 1–bytes.

#### **return\_code**

This is a pointer to a 4-byte field where the return code is to be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmstpn` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- `CM_OK`
- `CM_PRODUCT_SPECIFIC_ERROR`
- `CM_PROGRAM_PARAMETER_CHECK` — Either **conversation\_ID** or **TP\_name\_length** is invalid.
- `CM_PROGRAM_STATE_CHECK` — The program is not in initialize state.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in initialize state.
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cminit` function.
- This function changes the value of the *TP\_name* and *TP\_name\_length* characteristics.
- This function causes no state change.
- If the new TP name is invalid, it is not detected until the `ALLOCATE` is attempted.
- If **return\_code** is not `CM_OK`, the *TP\_name* and *TP\_name\_length* characteristics remain unchanged.

### Examples

The following example changes the *TP\_name* and *TP\_name\_length* characteristics.

```
#include <tpfmap.h>

    unsigned char convid[8];
    unsigned char name[64];
    signed int    len;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from initialize */
    .
    .
cmstp(convid,name,&len,&rcode);
    .          /* normal processing path          */
    .
    .
```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmalloc—Allocate a Conversation” on page 800
- “cminit—Initialize a Conversation” on page 820.

## cmtrts—Test for Request\_To\_Send Notification

This function determines if a REQUEST\_TO\_SEND notification was received from the remote program for the specified conversation.

### Format

```
#include <tpfmap.h>
void cmtrts(unsigned char *conversation_ID,
            signed int *request_to_send_received,
            signed int *return_code);
```

#### conversation\_ID

This is a pointer to an 8-byte character array that contains the conversation ID. This conversation ID must be the ID returned by the `cminit` (INITIALIZE) or `cmaccp` (ACCEPT\_CONVERSATION) that started this conversation.

#### request\_to\_send\_received

This is a pointer to a 4-byte field where a REQUEST\_TO\_SEND notification is placed. Possible values are:

##### CM\_REQ\_TO\_SEND\_RECEIVED

This value indicates that the local program received a REQUEST\_TO\_SEND notification from the remote program. This is a request that the local program enter receive state, which allows the remote program to enter send state.

##### CM\_REQ\_TO\_SEND\_NOT\_RECEIVED

This value indicates that a REQUEST\_TO\_SEND notification was not received.

If **return\_code** is not CM\_OK, **request\_to\_send\_received** does not contain a meaningful value.

#### return\_code

This is a pointer to a 4-byte field where the return code will be placed.

### Return Codes

The following is a list of return codes that can be returned to the program that called the `cmtrts` function. A complete list of the return codes and their definitions can be found in Table 45 on page 796.

- CM\_OK
- CM\_PRODUCT\_SPECIFIC\_ERROR
- CM\_PROGRAM\_PARAMETER\_CHECK — The specified **conversation\_ID** is invalid.
- CM\_PROGRAM\_STATE\_CHECK — The program is not in send, send-pending, or receive state.

### Programming Considerations

- You can execute this function on any I-stream.
- The conversation must be in one of the following states:  
receive  
send  
send-pending
- The value supplied in **conversation\_ID** must be the conversation ID returned by the `cmaccp` or `cminit` function.
- There are no characteristics that affect this function.

- This function does not cause a state change.
- When the local LU receives a REQUEST\_TO\_SEND notification, it holds the notification until the local program does a function with the **request\_to\_send\_received** parameter (such as this one). Only one notification is held at one time. Additional notifications are discarded until the held notification is sent to the local program.
- The local LU discards the notification after it is given to the local program.
- To enter receive state, the local transaction program must call the appropriate function, such as cmrcv or cmptr, when the REQUEST\_TO\_SEND indication is received.

## Examples

The following example checks to see if a REQUEST\_TO\_SEND notification was received.

```
#include <tpfmap.h>
```

```

    unsigned char *buff;
    signed int    rtsr;
    signed int    rcode;
    .
    .
/* set conversation_ID with value returned from accept or initialize */
    .
    .
cmtrts(convid,&rtsr,&rcode);
    .                /* normal processing path                */
    .
    .

```

## Related Information

- “Return Codes for Mapped Conversation Functions” on page 796
- “cmrts—Request Change to Send State” on page 829.

**cmtrts**



---

## TPF Collection Support: Environment Functions

TPF collection support (TPFCS) permits database access through an application program that informs the TPFCS database that you want to access the database. The TPFCS database assigns a token and saves any information required by the database to communicate with you. Once you are known to the TPFCS database, you have access to all of the collections in the database. The TPFCS database has no service that restricts your access to a subset of collections. If you are allowed to access the database, the TPFCS database then allows access to all the collections in the database.

---

### Type Definitions

TPFCS uses the following type definitions:

#### **TO2\_ENV\_PTR**

This type is defined as a pointer to void. This pointer value is set by calling the `T02_createEnv` function and passing a pointer to this pointer. It is set by TPFCS to point to the environment block.

#### **TO2\_PID**

This type defines the TPFCS persistent identifier (PID) that is assigned to a collection when it is created and is then used to refer to the collection until it is deleted from the TPFCS database.

#### **TO2\_PID\_PTR**

This type is defined as a `TO2_PID` pointer and should be set to point to a `TO2_PID` type.

#### **TO2\_BUF\_HDR**

This type defines the returned TPFCS data buffer header that is returned on an element retrieval using such TPFCS functions as `T02_at`, `T02_atKey`, `T02_atCursor`, and `T02_key`.

The structure of this buffer has five fields:

Field	Description
<b>spare</b>	Type long, reserved for IBM use.
<b>updateSeqNbr</b>	Type long, update sequence counter value.
<b>dataL</b>	Type long, length of the data.
<b>spare</b>	Type long, reserved for IBM use.
<b>data</b>	Array of ' ', the beginning of the actual data. The data does not contain the key.

The actual data element then follows the header.

#### **TO2\_BUF\_PTR**

This type is defined as a `TO2_BUF_HDR` pointer value.

<b>T02_ERR_CODE</b>	This type is defined as a long integer that, on return from a T02_getErrorCode function, contains the actual error code for the request. The error code returned is the error code stored in the environment block.
<b>T02_ERR_TEXT_PTR</b>	This type is defined as a character pointer and is returned from a T02_getErrorText call. It will point to text that describes the actual error that occurred.

---

## Retrieving Error Information

Use the following steps to retrieve error information:

1. Enter a request; for example:

```
T02_copyCollectionTemp
```

An error occurs and a zero is returned.

**Note:** If no error is found, a positive value is returned.

2. Enter the following to retrieve the error code value:

```
T02_getErrorCode
```

3. Enter the following to retrieve the error code message text for the returned value:

```
T02_getErrorText
```

For more information about error codes, see Table 46 on page 862.

## Boolean Error Handling

By convention, TPF API return codes indicate successful or unsuccessful completion by returning positive or zero values, respectively. This convention is violated somewhat by TPFCS API functions returning positive or zero values for TRUE or FALSE results, respectively.

To distinguish an unsuccessful API return code from a Boolean zero result, use the T02\_getErrorCode function call for error value retrieval. A zero error code value indicates a successful return of FALSE while a nonzero error code value indicates an error return from the Boolean API function. This applies for the following APIs:

```
T02_atEnd
T02_atLast
T02_more
T02_includes
T02_isEmpty
T02_isCollection.
```

**Note:** The error code in the environment block is not reset until another error occurs or a Boolean-type request is entered that returns FALSE.

## Error Handling

When an error occurs in the attempt to process a function, TPFCS sets an error code in the environment block and returns a 0 to the application program. For example:

```
#include <c$to2.h>          /* Needed for T02 API Functions */
#include <stdio.h>          /* APIs for standard I/O functions */
```

```

T02_PID          cursor;
T02_ENV_PTR      env_ptr;
T02_ERR_CODE     err_code; /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
:
/*****
/* Are there more elements after the current one? */
*****/
if (T02_more(&cursor, env_ptr) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_IS_FALSE)
    {
        printf("T02_more failed!\n");
        err_code = T02_getErrorCode(env_ptr);
        err_text_ptr = T02_getErrorText(env_ptr, err_code);
        printf("err_text_ptr is %s\n", err_text_ptr);
    }
    else
        printf("There are no more elements after the current element.\n");
}
else
    printf("There are more elements after the current element.\n");

```

---

## Error Code Summary

The following table contains a list of the TPF collection support (TPFCS) error codes with descriptions and user actions. Not all return codes are possible for every function. The individual TPFCS functions document the most common return codes for that function.

Table 46. Error Code Summary

Value	Name of Error	Description	User Action
1	TO2_SUCCESS TO2_IS_TRUE	TO2_IS_TRUE	This is not an error; it indicates a return value of TO2_IS_TRUE from the original function call.
0	TO2_ERROR TO2_IS_FALSE	TO2_IS_FALSE	This is not an error; it indicates a return value of TO2_IS_FALSE from the original function call.
-1	TO2_ERROR_METHOD	The application programming interface (API) or internal method is not known or is not supported for the collection type.	Use a valid API for the collection (see Table 47 on page 881).
-2	TO2_ERROR_ENV	The environment pointer passed to a TPFCS API is not valid.	Check the program logic.
-3	TO2_ERROR_USER	The user token is zero or not valid.	Enter a valid user token.
-4	TO2_ERROR_DATA_LGH	The data length is too large or is not valid.	Enter a valid data length.
-5	TO2_ERROR_LOCATOR_LGH	The locator (key/value) length is too large or is not valid.	Enter a valid key or value length.
-6	TO2_ERROR_SEQCTR	There was an update sequence counter mismatch. When an update is attempted, the update sequence counter that is passed with the update must match the current update sequence counter maintained with the collection.	Retrieve the element again for a successful update to take place and repeat the update request.
-7	TO2_ERROR_EXCEED_MAX_INDEX	There was an attempt to add an entry beyond the maximum entry.	Add the entry again ensuring it is not beyond the defined maximum entry value.
-8	TO2_ERROR_INDEX	The index is not valid. The value is 0 or negative.	Enter a valid index entry.
-9	TO2_ERROR_UNDEFINED_RECORD_TYPE	The specified record type is not known by FACS.	Enter a valid record type that is known by FACS or define it to FACS.
-10	TO2_ERROR_EODAD	Access was attempted beyond the end of the data or was for an empty collection.	This is a normal response when you reach the end of a collection while iterating through it.
-11	TO2_ERROR_CURSOR	The cursor is not valid; positioning is required.	Enter a valid cursor positioning request. The cursor positioning may have been lost because of an update activity for this collection.
-12	TO2_ERROR_EMPTY	Access was attempted to an empty collection.	None. This is a normal response.
-13	TO2_ERROR_RECORD_SIZE	The specified record type is not defined as 4K.	Define the record type as 4K or specify a different record type and try again.
-14	TO2_ERROR_CLASS_NOT_FOUND	The specified class name was not found.	Enter a valid class name.
-15	TO2_ERROR_PID	The persistent identifier (PID) or a pointer is not known or is not valid.	Use a valid PID.
-16	TO2_ERROR_PARAMETER	The parameter passed is not valid.	Use a valid parameter.

Table 46. Error Code Summary (continued)

Value	Name of Error	Description	User Action
-17	TO2_ERROR_DBID	The database name is not defined or is not valid.	Enter a database name that is defined or that is valid.
-18	TO2_ERROR_USERTKN	The user token is not defined or is not valid.	Enter a user token that is defined or that is valid.
-19	TO2_ERROR_NOT_INIT	TPFCS is not initialized.	Initialize TPFCS using the ZOODB INIT command.
-20	TO2_ERROR_ACCESS_MISMATCH	An incorrect mode was passed to a TO2_atRBAPut call.	Check the logic in the application.
-21	TO2_ERROR_ZERO_PID	The TPFCS PID is not valid. The PID is zero.	Enter a valid PID.
-22	TO2_ERROR_LOCATOR_NOT_UNIQUE	The TPFCS locator (key/value) is not unique.	Enter a unique key or value.
-23	TO2_ERROR_LOCATOR_NOT_FOUND	The TPFCS locator (key/value) is not found.	Enter a valid key or value.
-24	TO2_ERROR_RECID_DEF	The record ID is not defined or there is a definition in error.	Define the record ID or determine if there is an error in the specified definition.
-25	TO2_ERROR_BUFFER_SIZE	The passed buffer length is either too small (<20 bytes) or is negative.	Use a valid buffer length.
-28	TO2_ERROR_IO_ERROR_ON_ACCESS	There was a TPFCS input/output (I/O) error accessing the collection. TPFCS was unable to access the collection because of an irrecoverable I/O error. A SNAP dump was taken already to inform coverage programmers of the problem.	None.
-29	TO2_ERROR_PROPERTY_TYPE	The TPFCS property type that was specified was not valid.	Specify a valid property type.
-30	TO2_ERROR_PROPERTY_MODE	The TPFCS property mode that was specified was not valid.	Use a valid property mode.
-31	TO2_ERROR_MODE_MISMATCH	This error occurs when there is a property mode mismatch. There was an attempt to change a read-only property or to delete a property that cannot be deleted.	Check the program logic.
-32	TO2_ERROR_DELETED_PID	The persistent identifier (PID) is marked for deletion.	Check the program logic. You should not be attempting to access a deleted PID. Check to see if the delete was valid or if the PID was deleted by mistake.
-33	TO2_ERROR_UPDATE_NOT_ALLOWED_GFS	This is a TPFCS error that occurs when there is an attempt to update a collection when the get file storage (GFS) of the data store is not active.	Activate GFS by cycling the system to NORM state and try again.
-34	TO2_ERROR_IO_RECID	There was a record ID (RECID) error in the input/output (I/O) processing code. TPFCS was unable to access the collection because of an irrecoverable I/O error. A SNAP dump was taken already to inform coverage programmers of the problem.	None.

Table 46. Error Code Summary (continued)

Value	Name of Error	Description	User Action
-35	TO2_ERROR_IO_RCC	There was a record code check (RCC) error in the input/output (I/O) processing code. TPFCS was unable to access the collection because of an irrecoverable I/O error. A SNAP dump was taken to inform coverage programmers of the problem.	None.
-36	TO2_ERROR_IO_HARDWARE	There was a record code check (RCC) error in the I/O processing code. TPFCS was unable to access the collection because of an irrecoverable I/O error. A SNAP dump was taken to inform coverage programmers of the problem.	None.
-37	TO2_ERROR_DD_NOT_FOUND	The specified data definition (DD) name was not defined.	Specify a DD name that is defined.
-38	TO2_ERROR_NO_PROPERTY_DEFINED	There are no properties defined for the collection.	Define properties for the collection, if desired.
-39	TO2_ERROR_PROPERTY_NOT_DEFINED	A TPFCS property is not defined for the PID.	Define the property for the PID or specify a defined property.
-40	TO2_ERROR_RECID_DEF_MISMATCH	There was a pool-type mismatch between assigned record IDs.	Correct the pool-type mismatch and try again.
-41	TO2_ERROR_NOT_KEYED	There was an attempt to add all the elements from the source collection to the target collection for a nonkeyed collection using the T02_addAllFrom function.	Check to be sure that the collection is a keyed collection and try again.
-42	TO2_ERROR_INITIALIZED	There was an attempt to reinitialize TPF collection support and it is already initialized.	None.
-43	TO2_ERROR_ADD_SAME_PID	There was an attempt to add all the elements from the source collection to the target collection using the T02_addAllFrom function. The source and target have the same persistent identifier (PID).	Check the program logic. The target and source cannot have the same PID.
-48	TO2_ERROR_RECONSTRUCT_CHAIN	There was an unrecognized chain given for reconstruction.	Check the chain and try again.
-49	TO2_ERROR_INDEX_EXISTS	There was an attempt to redefine an existing recoup index.	Either delete the existing index and redefine it or, modify the existing index.
-50	TO2_ERROR_INDEX_UNKNOWN	There was an attempt to access a recoup index that is not defined.	Define the recoup index name or specify a defined one and try again.
-51	TO2_ERROR_OPTION_CODE	The passed option code in the list is not defined.	Define the option code and try again.
-52	TO2_ERROR_OPTION_LIST	No list end is found or there is an option list pointer that is not valid; an internal error occurred.	Do one of the following: <ul style="list-style-type: none"> <li>• Enter a valid option list pointer and try again.</li> <li>• Specify a list end and try again.</li> <li>• Contact your IBM service representative.</li> </ul>

Table 46. Error Code Summary (continued)

Value	Name of Error	Description	User Action
-53	TO2_ERROR_OPTION_VALUE	The passed option list value pointer is not valid.	Enter a valid option list value pointer.
-54	TO2_ERROR_LIST_TYPE	There was an incorrect list type code passed on the T02_createOptionList function.	Enter a correct list type code for the T02_createOptionList function.
-55	TO2_ERROR_IBMM4	There was an error accessing the required IBMM4 ordinals.	Determine the cause of the error and try again.
-56	TO2_ERROR_NOTBSS	A ZOODB INIT command request was issued in a subsystem other than the basic subsystem (BSS). The ZOODB INIT command is only supported when issued in the BSS.	Enter the ZOODB INIT request again in the BSS.
-57	TO2_ERROR_RESTORE_PID_MISMATCH	The persistent identifier (PID) of the restored collection does not match the input PID.	Check to be sure the PID of the input and restored collections match.
-58	TO2_ERROR_TIMEOUT	The device mount request timed out. The request was cancelled.	Try the request again.
-59	TO2_ERROR_TAPE_FORMAT	This error code is returned on restores when the record read does not match the expected record.	Check the tape that is mounted to be sure it is the right tape. If it is not the correct tape, mount the correct tape. If the tape is the right tape, report the problem to your service representative.
-60	TO2_ERROR_DD_NAME_EXISTS	The data definition (DD) name specified already exists.	Use a unique DD name.
-61	TO2_ERROR_PID_NOT_DELETED	The persistent identifier (PID) to reclaim is not marked for deletion.	Mark the PID to be reclaimed for deletion and try again.
-62	TO2_ERROR_OPTION_CONFLICT	This error occurs when an option on an option list either conflicts with another option or with an existing definition.	Determine why there is a conflict with an option or with an existing definition and try again.
-63	TO2_ERROR_NO_XTERNAL_DEVICES	There are no external devices currently available.	Wait until the external devices that are in use are available or allocate more external devices. In either case, the application that was attempting to use the external device must reissue the request that needed the device.
-64	TO2_ERROR_NO_XTERNAL_DEFINED	There are no external devices defined to the system.	Allocate one or more external devices and reissue the request.
-65	TO2_ERROR_PERMANENT_XTERNAL	There is a permanent hardware error when an attempt was made to use an external device.	The device should be examined to determine the cause of the error.
-66	TO2_ERROR_ADD_FROM_BLOB	There was an attempt to add all from a BLOB.	You can only copy a BLOB to another BLOB or copy another collection to a BLOB. Enter the request again.
-67	TO2_ERROR_RECOUP_ABORT	The recoup abort flag was found set. This error code is only returned for a T02_recoupCollection function call.	None.

Table 46. Error Code Summary (continued)

Value	Name of Error	Description	User Action
-68	TO2_ERROR_DEADLOCK	A deadlock condition has been detected.	Analyze the locking protocols used to access the collection that received the deadlock error by looking for an access path that does not follow the same locking protocol as the current program. Once the path has been determined, you must determine which protocol is correct and make all the accesses to the collection follow the same locking protocol. If the locking protocol of the application is correct, see your system programmer to make sure that there is no underlying locking problem in TPF collection support.
-69	TO2_ERROR_NODELETE	The specified access mode of the persistent identifier (PID) has been set to not allow deletion.	To delete the PID, change the access mode to allow deletion.
-70	TO2_ERROR_STORAGE	The TO2_getBL0B function was not able to allocate a large enough malloc storage area to hold the specified PID.	The PID will have to be read using either the TO2_getBL0BWithBuffer function or in parts by using the TO2_atRBA function.
-71	TO2_ERROR_KEYPATH_BUILD_ACTIVE	The process of creating the specified key path has not completed successfully. Therefore, the key path is not usable.	Issue the TO2_setKeyPath request again after a delay to allow the build process to end.
-72	TO2_ERROR_CORRUPTED_COLLECTION	The object part in the control record has a corrupted object identifier (ID).	Report this error to your system programmer who may be able to correct the problem by restoring the collection or patching the corrupted control record.
-73	TO2_ERROR_MAX_KEYPATHS	The maximum number of key paths for this collection has been reached.	Delete an old key path first and then add the new key path.
-74	TO2_ERROR_RESERVED_NAME	The name you specified for the function is a TPFCS reserved name.	Enter the function again and specify another name.
-75	TO2_ERROR_GFS	TPFCS processing could not be completed because get file storage (GFS) is not active.	Cycle the system to CRAS state or above and repeat the request.
-76	TO2_ERROR_DS_INUSE	An attempt was made to delete, migrate, or re-create a data store by an entry control block (ECB) that is using the data store.	Repeat the request after this ECB deletes all environments associated with this data store.
-77	TO2_ERROR_SUBSYSTEM	An attempt was made to delete, migrate, or re-create a data store from a subsystem other than the owning subsystem.	Repeat the request from the owning subsystem, which can be determined by entering the ZOODB DISPLAY command with the DS parameter specified.
-100	TO2_ERROR_NO_ROOM	There is no room in malloc storage to hold the requested collection.	Expand storage and try again.
-101	TO2_ERROR_UNASSIGNED_RRN	The relative record number (RRN) you requested is unassigned.	Assign a RRN and try again.
-102	TO2_ERROR_RRN_ASSIGNED	The RRN to be assigned is already assigned.	Assign a RRN that isn't already assigned and try again.



Table 46. Error Code Summary (continued)

Value	Name of Error	Description	User Action
-103	TO2_ERROR_REREAD	There was a read sequencing error. The collection must be read again.	Reissue the request after a short delay. If the request still fails, assume a hard error and take appropriate action for the application being run.
-104	TO2_ERROR_RRN_CHANGED	The RRN directory entry value has changed.	Determine the correct RRN and try again.
-105	TO2_ERROR_REREAD_PREVIOUS	There was a previous read sequencing error.	Enter the request again.
-200	TO2_ERROR_LOGIC_COLLECT	There is a logic error in the collection processing code.	Check the program logic and try again.
-210	TO2_ERROR_LOGIC_CURSOR	There is a logic error in the cursor processing code.	Check the program logic and try again.
-220	TO2_ERROR_LOGIC_IO	There is a logic error in the input/output (I/O) processing code.	Check the program logic and try again.
-221	TO2_ERROR_NO_SHADOW	There was a logic error when an attempt was made to read a shadow that does not exist.	Check the program logic and try again.
-230	TO2_ERROR_LOGIC_STRUCT	There is a logic error in the STRUCTURE processing code.	Check the program logic and try again.
-240	TO2_ERROR_LOGIC_DIRECT	There is a logic error in the DIRECTORY processing code.	Try the request again and if the error persists, consider the collection unavailable. TPFCS will dump on any logic error code.
-250	TO2_ERROR_LOGIC_MALOC	The requested malloc storage was unable to be allocated.	None. TPFCS will exit the entry control block (ECB) if it is unable to allocate maloc storage for its processing requirement.
-300	TO2_ERROR_XTERNAL	This is used to offset external device support errors.	None.
-301	TO2xd_ERROR_archiveUnavail	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-302	TO2xd_ERROR_levelInUse	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.

Table 46. Error Code Summary (continued)

Value	Name of Error	Description	User Action
-303	TO2xd_ERROR_maxArchive	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code.	None.
-304	TO2xd_ERROR_notOpen	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-305	TO2xd_ERROR_noDDN	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-306	TO2xd_ERROR_noPosition	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-307	TO2xd_ERROR_positionIncorrect	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.

Table 46. Error Code Summary (continued)

Value	Name of Error	Description	User Action
-308	TO2xd_ERROR_readOnly	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-309	TO2xd_ERROR_recordTooLong	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-310	TO2xd_ERROR_recordTooShort	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-311	TO2xd_ERROR_timeOut	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code.	None.
-312	TO2xd_ERROR_VOLSERmismatch	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.

Table 46. Error Code Summary (continued)

Value	Name of Error	Description	User Action
-313	TO2xd_ERROR_EOVwarning	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-314	TO2xd_ERROR_HWerror	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-315	TO2xd_ERROR_SDAunavail	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-316	TO2xd_ERROR_SDAwarning	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-317	TO2xd_ERROR_VOLSERinUse	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.

Table 46. Error Code Summary (continued)

Value	Name of Error	Description	User Action
-318	TO2xd_ERROR_noToken	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-319	TO2xd_ERROR_unableToPosition	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-320	TO2xd_ERROR_writeOnly	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-321	TO2xd_ERROR_EOError	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-322	TO2xd_ERROR_alreadyOpen	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.

Table 46. Error Code Summary (continued)

Value	Name of Error	Description	User Action
-323	TO2xd_ERROR_noBlock	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-324	TO2xd_ERROR_notBaseSSU	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.
-325	TO2xd_ERROR_noARCHIVEgroup	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code.	None.
-326	TO2xd_ERROR_noArchiveSDA	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code.	None.
-327	TO2xd_ERROR_libraryNotFound	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code.	None.
-328	TO2xd_ERROR_tapeBlocked	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.

Table 46. Error Code Summary (continued)

Value	Name of Error	Description	User Action
-329	T02xd_ERROR_logic	This is used to map the TPFxd_error numbers out of the TPFCS error numbers. This error code is only returned on a T02_capture, T02_restore, T02_restoreAsTemp, or T02_restoreWithOptions request. To convert a TPFCS error code to a TPFxd_error code, subtract 300 from the TPFCS error code. See "External Device Support" on page 1357 for more information.	None.

## TO2\_createEnv—Create an Environment Block

This function is called to create a temporary context block called the environment block. It is used to contain information about the user and the data store (DS) and to return error information. The environment block is required on every call to the database.

The TO2\_createEnv function will fill in the fields in the environment block that will be needed by TPFCS to satisfy any request.

### Format

```
#include <c$to2.h>
long TO2_createEnv (      TO2_ENV_PTR *env_ptr,
                          long         *userTokenPtr,
                          const char   *applID,
                          const char   *dsName);
```

#### env\_ptr

The pointer to a long integer, which is updated to contain a pointer to the created environment. This pointer must be defined as type TO2\_ENV\_PTR.

#### userTokenPtr

Reserved for future IBM use. This must point to a word that contains zeros.

#### applID

The pointer to a 32-character field containing the application name in EBCDIC, left-justified and padded with blanks.

#### dsName

The pointer to an 8-character field containing the data store name in EBCDIC, left-justified and padded with blanks.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DBID
TO2_ERROR_NOT_INIT
TO2_ERROR_USERTKN
```

### Programming Considerations

- Delete the created environment block by entering the TO2\_deleteEnv function.
- This function does not use TPF transaction services on behalf of the caller.
- Make sure a pointer to the environment pointer is passed as a parameter to this function. All other application programming interfaces (APIs) that use the environment require a pointer to the environment, not a pointer to the environment pointer.
- If the TO2\_createEnv function fails, a storage block will be returned to the application with the error code stored in it (as long as there is malloc space).



available). When this occurs, the T02\_createEnv function will work properly on the returned block. If no malloc storage is available, the entry control block (ECB) will exit with a dump.

## Examples

The following example creates a TPFCS environment to allow access to the named data store.

```
#include <c$to2.h>           /* T02 API function prototypes */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;     /* T02 environment pointer */
T02_USER_TOKEN userToken = 0; /* User token not used */

char applID[32] = "Test.Application.ID";
char dsname[8]  = "TEST.DB ";

T02_ERR_CODE    err_code;    /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
:
if (T02_createEnv(&env_ptr,
                 &userToken,
                 applID,
                 dsname) == T02_ERROR)
{
    printf("createEnv failed!\n");
    if (env_ptr != NULL)
        process_error(env_ptr);
}
else
    printf("T02_createEnv is successful!\n");
:
T02_deleteEnv (env_ptr);
```

## Related Information

"T02\_deleteEnv—Delete an Environment Block" on page 876.

## T02\_deleteEnv—Delete an Environment Block

This function is called to delete the environment block and the environment created when the T02\_createEnv function was called.

### Format

```
#include <c$to2.h>
long T02_deleteEnv (T02_ENV_PTR env_ptr);
```

#### env\_ptr

The pointer to the environment block that will be deleted.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

T02\_ERROR\_ENV

## Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example deletes a TPFCS user environment.

```
#include <c$to2.h>           /* T02 API function prototypes */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;     /* T02 environment pointer */
:
:
if (T02_deleteEnv(env_ptr) == T02_ERROR)

    printf("deleteEnv failed!\n");
else
    printf("deleteEnv is successful!\n");
```

### Related Information

“T02\_createEnv—Create an Environment Block” on page 874.

## TO2\_getErrorCode—Retrieve the Error Code Value

This function retrieves the error code value from the environment after a reported TPFCS error.

### Format

```
#include <c$to2.h>
TO2_ERR_CODE TO2_getErrorCode (const TO2_ENV_PTR env_ptr);
```

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

### Normal Return

The normal return is the error code from the environment.

### Error Return

Not applicable.

## Programming Considerations

- By convention, application programming interface (API) return codes indicate successful or unsuccessful completion by returning positive or zero values respectively. This convention is violated somewhat by API functions returning positive or zero values for TO2\_IS\_TRUE or TO2\_IS\_FALSE results. To distinguish an unsuccessful API return code from a Boolean FALSE result, use the TO2\_getErrorCode function to retrieve any stored return code value in the user environment. A TO2\_IS\_FALSE error code value indicates a successful return while a negative error code value indicates an error return from the Boolean API function.
- An error in the environment is not reset until another error or a TO2\_IS\_FALSE condition occurs.

## Examples

The following example retrieves the error code value from a TPFCS environment.

```
#include <c$to2.h>                /* Needed for TO2 API Functions */
#include <stdio.h>                 /* APIs for standard I/O functions */

TO2_PID      cursor;
TO2_ENV_PTR  env_ptr;
TO2_ERR_CODE err_code;           /* TO2 error code value */
TO2_ERR_TEXT_PTR err_text_ptr;  /* TO2 error code text pointer */
:
:
/*****
/* Increment cursor to point to next element in the collection. */
*****/
if (TO2_cursorPlus(&cursor, env_ptr) == TO2_ERROR)
{
    err_code = TO2_getErrorCode(env_ptr);
    if (err_code != TO2_ERROR_EODAD)
    {
        printf("TO2_cursorPlus failed!\n");
        process_error(env_ptr);
    }
    else
        printf("Cursor is already positioned at the last element.\n");
}
else
    printf("TO2_atCursorPlus successful!\n");
```

## T02\_getErrorCode

The following example tests to see if the cursor is pointing at the end of the collection.

```
#include <c$to2.h>                /* Needed for T02 API Functions    */
#include <stdio.h>

T02_PID      cursor;
T02_ENV_PTR  env_ptr;
T02_ERR_CODE err_code;    /* T02 error code value          */
:
:
/*****
/* Is the cursor pointing to the end of the collection?          */
*****/
if (T02_atEnd(&cursor, env_ptr)) == T02_IS_FALSE)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_IS_FALSE)
    {
        printf("T02_atEnd failed!\n");
        process_error(env_ptr);
    }
    else
        printf("Cursor is not pointing to the end of the collection.\n");
}
else
    printf("Cursor is pointing to the end of the collection.\n");
```

## Related Information

“T02\_getErrorText–Retrieve the Associated Error Text” on page 879.

## T02\_getErrorText—Retrieve the Associated Error Text

This function retrieves the associated error text for a TPFCS error code.

### Format

```
#include <c$to2.h>
T02_ERR_TEXT_PTR T02_getErrorText (const T02_ENV_PTR env_ptr,
                                   const T02_ERR_CODE err_code);
```

#### **env\_ptr**

The pointer to the environment used on the call that returned the error.

#### **err\_code**

The field that contains the error code value returned by the T02\_getErrorCode function.

### Normal Return

The normal return is a pointer to a NULL delimited text string. The character value at pointer address `-1` is the length of the text string not including the NULL delimiter.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example retrieves the error code text associated with an error code value returned by TPFCS.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;      /* PTR to the T02 environment */
T02_PID       collect;      /* will hold collection's PID */
T02_ERR_CODE   err_code;    /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
long          entryLength=100; /* set to entry length */
:
:
if (T02_createArray(&collect,
                  env_ptr,
                  &entryLength) == T02_ERROR)
{
    printf("T02_createArray failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
    printf("T02_createArray successful!\n");
```

### Related Information

"T02\_getErrorCode—Retrieve the Error Code Value" on page 877.

**TO2\_getErrorText**

## TPF Collection Support: Non-Cursor

This chapter provides information about TPF collection support (TPFCS) for non-cursor collection functions.

A *collection* is an abstract arrangement of data that allows you to manage a group of data elements that have common attributes. Collections are used to store and manage elements. Different collections have different internal structures and different access functions for element storage and retrieval. For more information about collections, see *TPF Application Programming*.

### Collection Support for Non-Cursor APIs

Table 47 lists the supported collection classes for non-cursor APIs. An X indicates support for that class. Refer to the following key for collection class names.

#### Key

Symbol	Collection	Symbol	Collection	Symbol	Collection
ARR	Array	KS	Key Set	SB	Sorted Bag
BAG	Bag	KSB	Key Sorted Bag	SEQ	Sequence
BLB	BLOB	KSS	Key Sorted Set	SET	Set
KB	Key Bag	Log	Log	SS	Sorted Set
KL	Keyed Log				

Table 47. Collection Support: Non-Cursor APIs

API Name	ARR	BAG	BLB	KB	KL	KS	KSB	KSS	LOG	SB	SEQ	SET	SS
TO2_add	X	X	X		X				X	X	X	X	X
TO2_addAllFrom	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_addAtIndex											X		
TO2_addKeyPath				X		X	X	X		X			X
TO2_addRecoupIndexEntry	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_asSequenceCollection	X	X		X	X	X	X	X	X	X	X	X	X
TO2_associateRecoupIndexWithPID	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_asSortedCollection	X	X		X	X	X	X	X	X	X	X	X	X
TO2_at	X				X				X		X		
TO2_atKey				X	X	X	X	X					
TO2_atKeyPut						X		X					
TO2_atKeyWithBuffer				X	X	X	X	X					
TO2_atNewKeyPut				X		X	X	X					
TO2_atPut	X										X		
TO2_atRBA			X										
TO2_atRBAPut			X										
TO2_atRBAWithBuffer			X										
TO2_atWithBuffer	X				X				X		X		
TO2_capture	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_copyCollection	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_copyCollectionTemp	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_copyCollectionWithOptions	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_create...	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_create...Temp	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_createRecoupIndex	X	X	X	X	X	X	X	X	X	X	X	X	X

Table 47. Collection Support: Non-Cursor APIs (continued)

API Name	ARR	BAG	BLB	KB	KL	KS	KSB	KSS	LOG	SB	SEQ	SET	SS
TO2_create...WithOptions	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_deleteCollection	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_deletePID	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_deleteRecoupIndex	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_deleteRecoupIndexEntry	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_getBLOB			X										
TO2_getBLOBwithBuffer			X										
TO2_getCollectionAccessMode	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_getCollectionKeys				X	X	X	X	X					
TO2_getCollectionType	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_getDRprotect	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_getMaxDataLength	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_getMaxKeyLength				X	X	X	X	X					
TO2_getSortFieldValues										X			X
TO2_includes		X										X	
TO2_isCollection	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_makeEmpty	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_maxEntry					X				X				
TO2_removeIndex	X										X		
TO2_removeKey				X		X	X	X					
TO2_removeKeyPath				X		X	X	X		X			X
TO2_removeRBA			X										
TO2_removeRecoupIndexFromPID	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_removeValue		X										X	
TO2_removeValueAll		X										X	
TO2_replaceBLOB			X										
TO2_restore	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_restoreAsTemp	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_restoreWithOptions	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_setCollectionAccessMode	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_setDRprotect	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_setSize			X										
TO2_size	X	X	X	X	X	X	X	X	X	X	X	X	X
TO2_writeNewBLOB			X										



## T02\_add–Add an Element to a Collection

This function adds the specified data to the collection at the position appropriate for the type of collection being added:

Type	Position
<b>Array</b>	Appends the element to the current end of the collection.
<b>Bag</b>	Adds the element to the collection.
<b>BLOB</b>	Appends the data to the current end of the collection.
<b>Key Bag</b>	Not supported.
<b>Key Set</b>	Not supported.
<b>Key Sorted Bag</b>	Not supported.
<b>Key Sorted Set</b>	Not supported.
<b>Keyed Log</b>	Appends the element to the logical end of the collection and updates the secondary index with the key field value. The secondary index is a user-defined key field that can be used for searching.
<b>Log</b>	Appends the element to the logical end of the collection.
<b>Sequence</b>	Appends the element to the current end of the collection.
<b>Set</b>	Adds the element to the collection after checking to make sure the element is not a duplicate.
<b>Sorted Bag</b>	Adds the element to the collection in the correct sequence position. Elements with the same sort field values are added in a first-in-first-out (FIFO) order in the collection.
<b>Sorted Set</b>	Adds the element in sort sequence determined by the sort field.

## Format

```
#include <c$to2.h>
long T02_add (const T02_PID_PTR pid_ptr,
               T02_ENV_PTR env_ptr,
               const void *data,
               const long *dataLength);
```

### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection to which the element will be added.

### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### data

The pointer to the element that will be stored in the collection.

## T02\_add

### dataLength

The pointer to an area that contains the length of the element that will be stored.

## Normal Return

For a normal return for the following collections, T02\_add returns the position index of where the element was added:

- Array
- BLOB
- Keyed log
- Log
- Sequence.

**Note:** The position index for these collections is 1-based; that is, 1 represents the first element.

For a normal return for the following collections, T02\_add returns a positive value:

- Bag
- Set
- Sorted bag
- Sorted set.

**Note:** The set collection returns a positive success return for duplicate elements even though set does not allow duplicate elements.

## Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_UNIQUE
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- A commit scope will be created for the T02\_add request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_add request, the scope will be rolled back.
- For all collections, the T02\_add request skips the test of the update sequence counter, which checks for the expected value for the counter.

## Examples

The following example adds an item to a bag collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions*/
```

```

    T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
    T02_PID        bag;              /* will store PID after create */
    char           item[] = "Item A"; /* data */
    long           itemsiz;
    :
    :
/*****
/* Make sure that the bag is created before arriving here... */
*****/

itemsiz = sizeof(item);
if (T02_add(&bag,
           env_ptr,
           item,
           &itemsiz) == T02_ERROR)
{
    printf("T02_add failed!\n");
    process_error(env_ptr);
}

else
{
    printf("T02_add successful!\n");
}

```

## Related Information

- “TO2\_addAllFrom—Add All from Source Collection” on page 886
- “TO2\_at—Return the Specified Element by Index” on page 903
- “TO2\_atPut—Update the Specified Element” on page 913
- “TO2\_removeIndex—Remove Element from the Index” on page 1078
- “TO2\_removeKey—Remove the Key-Entry Pair” on page 1080
- “TO2\_removeRBA—Remove an Area from a BLOB” on page 1084
- “TO2\_removeValue—Remove Value” on page 1088
- “TO2\_removeValueAll—Remove All Matching Values from the Collection” on page 1090.

## T02\_addAllFrom—Add All from Source Collection

This function adds all the elements from the source collection to the target collection in the most correct way. Because this function uses either a `T02_add` or `T02_atNewKeyPut` function depending on the target collection type, all the rules for the two functions and the target collection are followed. For this function, the source and target do not have to be the same type.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_addAllFrom (const T02_PID_PTR  target_pid_ptr,
                    T02_ENV_PTR    env_ptr,
                    const T02_PID_PTR  source_pid_ptr);
```

#### target\_pid\_ptr

The pointer to the persistent identifier (PID) assigned to the target collection. The target collection must have been previously created. The target and source collections do not have to be the same collection type.

#### env\_ptr

The pointer to the environment as returned by the `T02_createEnv` function.

#### source\_pid\_ptr

The pointer to the PID of the source collection that has elements that will be added to the target collection. The source collection is unchanged.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the `T02_getErrorCode` function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- A commit scope will be created for the `T02_addAllFrom` request. If the request is successful, the scope will be committed. If an error occurs while processing the `T02_addAllFrom` request, the scope will be rolled back.
- See the `ZBROW COL` command (with the `ADDAll` parameter) in *TPF Operations* for more information about the rules for the `T02_add` and `T02_atNewKeyPut` C functions.

- If a duplicate element is attempted to be added to a unique collection, the attempt is ignored and processing continues. If any other error is encountered, processing will stop and the target collection will be left in its original state.

## Examples

The following example adds an item to a bag collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;      /* Pointer to the T02 environment */
T02_PID        bag, bag_temp; /* will hold bag PIDs */
:
:
/*****
/* Make sure the bags referenced by bag and bag_temp are
/*   created before arriving here....
/* Add all from bag to bag_temp
*****/
if (T02_addAllFrom(&bag_temp,
                  env_ptr,
                  &bag) == T02_ERROR)
{
    printf("T02_addAllFrom failed!\n");
    process_error(env_ptr);
}

else
{
    printf("T02_addAllFrom successful!\n");
}
```

## Related Information

- “T02\_add–Add an Element to a Collection” on page 883
- “T02\_copyCollection–Make a Persistent Copy of the Collection” on page 927
- “T02\_migrateCollection–Migrate a Collection” on page 1318.

## T02\_addAtIndex–Insert an Element in an Sequence Collection

This function inserts the specified data in the collection at the specified relative index. The T02\_addAtIndex function sets the update sequence counter of the element to zero.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <to2.h>
long T02_addAtIndex(const T02_PID_PTR  pid_ptr,
                    T02_ENV_PTR  env_ptr,
                    const long    *index,
                    const void    *data,
                    const long    *dataLength);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that is the target of the insertion.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### index

The pointer to a variable that contains the relative index at which the element will be inserted. The index is 1-based; that is, the first element in the collection has an index of 1.

#### data

The pointer to the element that will be stored in the collection.

#### dataLength

The pointer to an area that contains the length of the element that will be stored.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_EODAD
T02_ERROR_INDEX
T02_ERROR_METHOD
T02_ERROR_NOT_INIT
T02_ERROR_PID
T02_ERROR_UPDATE_NOT_ALLOWED
T02_ERROR_ZERO_PID
```

## Programming Considerations

Inserting an element into a sequence collection increases by one position the relative indexes of all elements following the inserted element. The T02\_addAtIndex function can only be used to insert an element at the beginning of the collection, between two already existing elements, or as the element immediately after the current last element.

## Examples

The following example adds an item to a sequence collection at the specified relative index.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
T02_PID        seqCol;           /* PID for the target collection */
char           item =[] "Item A"; /* data */
long           itemsiz;
long           index=5;
:
:
/*****
/* Assume that the ordered collection has already been created..
*****/

itemsiz = sizeof(item);
if (T02_addAtIndex(&seqCol,
                  env_ptr,
                  &index,
                  item,
                  &itemsiz) == T02_ERROR)
{
    printf("T02_addAtIndex failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_addAtIndex successful!\n");
}
```

## Related Information

- “T02\_at—Return the Specified Element by Index” on page 903
- “T02\_atPut—Update the Specified Element” on page 913
- “T02\_removeIndex—Remove Element from the Index” on page 1078.

## T02\_addKeyPath—Add a Key Path to a Collection

This function allows an application program to add key paths to a persistent collection to access the data elements of the collection in an order different from the primary key path. The application can define as many as 16 unique alternate key paths (in addition to the primary key path) for a collection.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_addKeyPath (const T02_PID_PTR pid_ptr,
                    T02_ENV_PTR env_ptr,
                    const char *name,
                    const long *fieldLength,
                    const long *fieldDisplacement);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### name

The pointer to a string that is the name that will be assigned to the new key path. The key path name can be a maximum of 16 characters. Two key paths in the same collection **cannot** have the same name.

#### fieldLength

The pointer to an area that contains the length of the key path field. The field length must be 248 bytes or less.

#### fieldDisplacement

The pointer to an area that contains the 0-based displacement of the key path field.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
T02_ERROR_LOCATOR_NOT_UNIQUE
T02_ERROR_MAX_KEYPATHS
T02_ERROR_METHOD
T02_ERROR_NOT_INIT
T02_ERROR_PID
T02_ERROR_RESERVED_NAME
T02_ERROR_ZERO_PID
```



## Programming Considerations

- Alternate key paths are not supported for temporary collections. If this application programming interface (API) is issued against a temporary collection, a TO2\_ERROR\_METHOD error code will be returned.
- If a data element is not longer than the displacement value for the field, the data element will not be included in the key path. If a data element is not long enough to contain the entire specified key path field, the amount available will be padded with zeros and used as the key for the key path.
- When key paths are added to a collection that already exists, an asynchronous task is run to build the new key path. Until the task has completed successfully, the key path cannot be used to access the collection. If an attempt is made to use the key path while it is being built, the TO2\_ERROR\_KEYPATH\_BUILD\_ACTIVE error code is returned to the caller.
- TO2\_PRIME\_KEYPATH is a reserved key path name and **cannot** be used for naming alternate key paths.
- Whenever a collection is updated, TPF collection support (TPFCS) updates the key paths as required.
- This function uses TPF transaction services on behalf of the caller.

## Examples

The following example creates key paths for the zip code, state, and country code fields of a customer entry for the passed collection pointer.

```
#include <c$to2.h>           /* T02 API function prototypes */
#include <stddef.h>          /* standard definitions (offsetof) */
#include <stdio.h>           /* APIs for standard I/O functions */

/* Customer name entry: */
struct customerEntry {
    char    lastName[48]; /* customer's last name */
    char    firstName[48]; /* customer's first name */
    char    street[48]; /* street address */
    char    city[48]; /* city name */
    long    stateCode;
    long    countryCode;
    long    zipCode; /* customer's zip code */
};
typedef struct customerEntry CUST;

long createCustomerKeyPath(T02_PID_PTR pid_ptr,
                          T02_ENV_PTR env_ptr)
{
    long rc; /* return code */
    /* key path field values */
    CUST customer;
    char name[17]; /* name of key path */
    long offset; /* off set of field */
    long length; /* length of field */

    /* work areas */

    /*****
    /* Using T02_addKeyPath create the additional key paths.
    *****/

    length=sizeof(customer.zipCode);
    offset=offsetof(customer, zipCode);
    strcpy(name,"ZIPCODE");
```

## TO2\_addAtIndex

```
if (TO2_addKeyPath(pid_ptr,
                  env_ptr,
                  name,
                  &length,
                  &offset) == T02_ERROR)
{
    printf("TO2_addKeyPath failed!\n");
    process_error(env_ptr);
}

length=sizeof(customer.stateCode);
offset=offsetof(customer, stateCode);
strcpy(name,"STATECODE");
if (TO2_addKeyPath(pid_ptr,
                  env_ptr,
                  name,
                  &length,
                  &offset) == T02_ERROR)
{
    printf("TO2_addKeyPath failed!\n");
    process_error(env_ptr);
}

length=sizeof(customer.countryCode);
offset=offsetof(customer, countryCode);
strcpy(name,"COUNTRYCODE");
if (TO2_addKeyPath(pid_ptr,
                  env_ptr,
                  name,
                  &length,
                  &offset) == T02_ERROR)
{
    printf("TO2_addKeyPath failed!\n");
    process_error(env_ptr);
}
/*      ... return to the caller ...      */
return 1;
}
```

## Related Information

- “TO2\_getCurrentKey—Retrieve the Current Key” on page 1141
- “TO2\_removeKeyPath—Remove a Key Path from a Collection” on page 1082
- “TO2\_setKeyPath—Set a Cursor to Use a Specific Key Path” on page 1180.

## TO2\_addRecoupIndexEntry—Add an Entry to a Recoup Index

This function describes the location of user-embedded file addresses or persistent identifiers (PIDs) in collection elements.

### Format

```
#include <c$to2.h>
long TO2_addRecoupIndexEntry (      T02_ENV_PTR          env_ptr,
                                   const void             *indexName,
                                   const void             *entryToken,
                                   const enum T02_RECOUP_ENTRY_TYPE entryType,
                                   const long             *displacement,
                                   const enum T02_RECOUP_ENTRY_ACCESS accessType,
                                   const long             *accessValueLen,
                                   const void             *accessValue);
```

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **indexName**

The pointer to the 8-byte recoup index name to which the entry will be added.

#### **entryToken**

The pointer to the user-defined entry token. The token is an identifier for the given entry and must be unique in an index and must be 8 bytes long.

#### **entryType**

The embedded recoup information type indicator.

#### **T02\_RECOUP\_ENTRY\_FA**

The index entry being added describes a 16-byte area that contains a file address. See *TPF Database Reference* for more information about the format of the file addresses embedded in collections.

#### **T02\_RECOUP\_ENTRY\_PID**

The index entry being added describes a PID.

#### **displacement**

The displacement into the collection element for the user-embedded PID or file address. The displacement is 0-based.

#### **accessType**

For heterogeneous collections, the means to access the collection element.

#### **T02\_RECOUP\_ACCESS\_INDEX**

Use an index to access the element.

#### **T02\_RECOUP\_ACCESS\_KEY**

Use a key to access the element.

#### **T02\_RECOUP\_ACCESS\_NOTUSED**

Use for homogeneous and binary large object (BLOB) collections.

#### **accessValueLen**

For heterogeneous collections, the length of the index or key value that is used to access the element that contains the embedded recoup information. The length of an index must be 4 bytes. For homogeneous and BLOB collections, NULL is passed.

#### **accessValue**

For heterogeneous collections, the index or key value that is used to access the element that contains the embedded recoup information. For homogeneous and BLOB collections, NULL is passed.

## T02\_addRecoupIndexEntry

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

T02\_ERROR\_ENV

### Programming Considerations

- For homogeneous collections, TPF collection support (TPFCS) recoup will iterate through the collection. The displacement to the embedded item is applied to each element and it begins at the start of the element.
- For a binary large object (BLOB), the displacement is from the start of the BLOB.
- This function uses TPF transaction services on behalf of the caller.

### Examples

The following example adds an entry to a recoup index.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

#define KEY_OFFSET 20

T02_ENV_PTR    env_ptr;          /* Pointer to T02 environment */
u_char        indexName[]="INDEX001"; /* index identifier */
char          entryToken[8];     /* pointer to the entry token */
T02_RECOUP_ENTRY_TYPE entryType; /* PID or FA index entry */
long          displacement;      /* displacement to element */
T02_RECOUP_ENTRY_ACCESS accessType; /* key or index to access element */
long          accessValueLen;    /* key length */
u_char        accessValue[]="JACOB"; /* key to access element */

:

memcpy(entryToken, "MYDSNAME",8);
entryType = T02_RECOUP_ENTRY_PID;
displacement = KEY_OFFSET;
accessType = T02_RECOUP_ACCESS_KEY;
accessValueLen = sizeof(accessValue);

if (T02_addRecoupIndexEntry(env_ptr,
                           indexName,
                           &entryToken,
                           entryType,
                           &displacement,
                           accessType,
                           &accessValueLen,
                           accessValue) == T02_ERROR)
{
    printf ("T02_addRecoupIndexEntry failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_addRecoupIndexEntry successful\n");
```

**Related Information**

- “TO2\_associateRecoupIndexWithPID—Create a PID to Index Association” on page 899
- “TO2\_createRecoupIndex—Create a Recoup Index” on page 998
- “TO2\_deleteRecoupIndexEntry—Delete an Entry from an Index” on page 1041.

## **T02\_asOrderedCollection–Return Contents As Ordered**

This function has been renamed to `T02_asSequenceCollection`. For more information, see “`T02_asSequenceCollection–Return Contents As Sequence`” on page 897.

## T02\_asSequenceCollection–Return Contents As Sequence

This function creates a temporary sequence collection that contains all the elements of the source collection. Duplicate elements are allowed, and the order of the sequence collection is random.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_asSequenceCollection (const T02_PID_PTR    source_pid_ptr,
                                T02_ENV_PTR        env_ptr,
                                T02_PID_PTR        sequenceCollectionPID);
```

**source\_pid\_ptr**

The pointer to the persistent identifier (PID) assigned to the source collection.

**env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

**sequenceCollectionPID**

The pointer to where the PID is stored for the temporary sequence collection on return.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_NOT_INIT
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the entry control block (ECB) exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- The keys of a keyed collection are not included in the collection elements. The T02\_asSequenceCollection function only looks at the elements of a keyed collection.

### Examples

The following example creates a temporary sequence collection containing all the elements of the source collection.

## T02\_asSequenceCollection

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;     /* Pointer to the T02 environment */
T02_PID        sourcePID;   /* holds PID of source collection */
T02_PID        sequencePID; /* will hold PID of sequence col. */
:
if (T02_asSequenceCollection(&sourcePID,
                             env_ptr,
                             &sequencePID) == T02_ERROR)
{
    printf("T02_asSequenceCollection failed!\n");
    process_error(env_ptr);
    return;
}
else
{
    printf("T02_asSequenceCollection successful!\n");
}
:
T02_deleteCollection(sequencePID, env_ptr);
```

## Related Information

- “T02\_addAllFrom—Add All from Source Collection” on page 886
- “T02\_asSortedCollection—Return Contents As Sorted Bag” on page 901
- “T02\_copyCollection—Make a Persistent Copy of the Collection” on page 927
- “T02\_deleteCollection—Delete a Collection” on page 1036.



## T02\_associateRecoupIndexWithPID—Create a PID to Index Association

This function associates a recoup index with a collection.

### Format

```
#include <c$to2.h>
long T02_associateRecoupIndexWithPID (const T02_PID_PTR  pid_ptr,
                                       T02_ENV_PTR    env_ptr,
                                       const void      *indexName);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### indexName

The pointer to the 8-byte recoup index name that will be assigned to the collection.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_PID
```

### Programming Considerations

- This function uses TPF transaction services on behalf of the caller.
- Even though a given recoup index may be associated with more than one collection simultaneously, a given collection can only be associated with one recoup index at any given time. On normal return from this function, if the collection was previously associated with another recoup index, that previous association is replaced with the new association.

### Examples

The following example assigns a recoup index to a collection.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_PID      collect;       /* Target object's PID */
T02_ENV_PTR  env_ptr;       /* Pointer to T02 environment */
u_char      * indexName;    /* index identifier */
:
:
if (T02_associateRecoupIndexWithPID(&collect,
                                     env_ptr,
                                     indexName) == T02_ERROR)
{
    printf ("T02_associateRecoupIndexWithPID failed!\n");
    process_error(env_ptr);
}
```

## TO2\_associateRecoupIndexWithPID

```
    }  
    else  
        printf("TO2_associateRecoupIndexWithPID successful!\n");
```

## Related Information

- “TO2\_addRecoupIndexEntry—Add an Entry to a Recoup Index” on page 893
- “TO2\_createRecoupIndex—Create a Recoup Index” on page 998
- “TO2\_removeRecoupIndexFromPID—Remove a PID to Index Association” on page 1086.

## TO2\_asSortedCollection–Return Contents As Sorted Bag

This function creates a temporary sorted bag collection containing all the elements of the source collection.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_asSortedCollection (const T02_PID_PTR    source_pid_ptr,
                              T02_ENV_PTR        env_ptr,
                              T02_PID_PTR        sortedCollectionPID,
                              const long         *sortFieldLength,
                              const long         *sortFieldDisplacement);
```

#### **source\_pid\_ptr**

The pointer to where the persistent identifier (PID) assigned to the source collection is stored.

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **sortedCollectionPID**

The pointer to where the PID for the temporary sorted bag collection will be returned.

#### **sortFieldLength**

The pointer to the length of the field in the source collection that will be used as the sort field. The maximum length of the sort field is 248 bytes and it must be less than or equal to the entry length.

#### **sortFieldDisplacement**

The pointer to the displacement in the source collection entry to the start of the sort field. The displacement is 0-based. The displacement of the sort field plus the length of the sort field must be less than or equal to the length of the source collection entry.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_DATA_LGH
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- The keys of a keyed collection are not included in the collection element. The T02\_asSortedCollection function only looks at the elements of a keyed collection.

## Examples

The following example creates a temporary sorted bag collection containing all the elements of the source collection.

```
#include <c$to2.h>           /* Needed for T02 API functions      */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;     /* Pointer to the T02 environment */
T02_PID        sourcePID;   /* holds PID of source collection */
T02_PID        sortedPID;   /* will hold PID of sorted bag col. */

long           fieldLength; /* should set to length of sort field */
long           fieldDisp;  /* should set to displacement within */
                /* entry of sort field              */
                /*
                :
if (T02_asSortedCollection(&sourcePID,
                           env_ptr,
                           &sortedPID,
                           &fieldLength,
                           &fieldDisp) == T02_ERROR)
{
    printf("T02_asSortedCollection failed!\n");
    process_error(env_ptr);
    return;
}
else
{
    printf("T02_asSortedCollection successful!\n");
}
                :
T02_deleteCollection(sortedPID, env_ptr);
```

## Related Information

- “T02\_addAllFrom—Add All from Source Collection” on page 886
- “T02\_asSequenceCollection—Return Contents As Sequence” on page 897
- “T02\_copyCollection—Make a Persistent Copy of the Collection” on page 927
- “T02\_deleteCollection—Delete a Collection” on page 1036.

## TO2\_at–Return the Specified Element by Index

This function returns the data at the specified position in the specified collection. The collection is not changed. If the collection is an array and the position is marked as empty, this method returns a data length of 0 and an update sequence counter of 0.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_at (const TO2_PID_PTR  pid_ptr,
                     TO2_ENV_PTR    env_ptr,
                     const long      *index);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) of the collection that will be accessed.

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### index

The pointer to the index of the entry that has contents that will be returned.

### Normal Return

The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a NULL pointer. When a NULL pointer is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_INDEX
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_REREAD
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- It is the responsibility of the caller to free the returned buffer once the caller has stopped using the buffer.
- This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example obtains a buffer with a copy of the third element of a specified collection.

## T02\_at

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_BUF_PTR    buffer_ptr;
T02_PID        collect;
T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
long           position=3;       /* We will try to get third element */
char           *dataField;
:
:
/*****
/* Copy the element at the specified position.
*****/
if ((buffer_ptr = T02_at(&collect,
                        env_ptr,
                        &position)) == NULL)
{
    printf("T02_at failed!\n");
    process_error(env_ptr);
    return;
}
else
{
    dataField = buffer_ptr->data;
    printf("T02_at successful!\n");
}
:
:
free(buffer_ptr);
```

## Related Information

- “T02\_add—Add an Element to a Collection” on page 883
- “T02\_atCursor—Return the Element Pointed to by the Cursor” on page 1119
- “T02\_atKey—Return the Specified Element by Key” on page 905
- “T02\_atPut—Update the Specified Element” on page 913
- “T02\_atWithBuffer—Retrieve an Element from a Collection” on page 922.

## TO2\_atKey—Return the Specified Element by Key

This function searches the specified collection for the specified key and, if found, returns the associated element value. If the collection contains duplicate keys equal to the specified key, the first one found will be returned.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_atKey (const TO2_PID_PTR pid_ptr,
                        TO2_ENV_PTR env_ptr,
                        const void *key,
                        const long *keyLength);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the value that will be used as the key to locate the specific entry.

#### keyLength

The pointer to the length of the key. The maximum length is 248 bytes. For key sorted set collections, the key length can be shorter than the defined key to allow a locate by partial key. The key comparison is always from the first character of the key for the specified length. For all other keyed collections, the key length must equal the maximum defined key length of the collection.

### Normal Return

The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a NULL pointer. When a NULL pointer is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_REREAD
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- It is the responsibility of the caller to free the returned buffer once the caller has stopped using the buffer.
- This function does not use TPF transaction services on behalf of the caller.

## T02\_atKey

### Examples

The following example searches a keyed collection to determine if the collection contains an element with the specified key. If an element with that key is found, a buffer is obtained with a copy of the element.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions*/

T02_BUF_PTR      elem_ptr;
T02_PID          keyb;           /* will set to KeyBag PID */
T02_ENV_PTR      env_ptr;       /* Pointer to the T02 environment */
char             keya[] = "KeyA"; /* key to search for */
long             keylength;
char             *dataField;
:
:
:
/*****
/* Test to see if collection contains entry for key in keya */
*****/

keylength = sizeof(keya);

if ((elem_ptr = T02_atKey(&keyb,
                        env_ptr,
                        keya,
                        &keylength)) == 0)
{
    printf("T02_atKey failed!\n");
    process_error(env_ptr);
    return;
}
else
{
    dataField = elem_ptr->data;
    printf("T02_atKey is successful. Key found!\n");
}
:
:
free(elem_ptr);
```

### Related Information

- “T02\_at–Return the Specified Element by Index” on page 903
- “T02\_atKeyPut–Update the Specified Element Using a Key” on page 907
- “T02\_atKeyWithBuffer–Return the Specified Element” on page 909
- “T02\_atNewKeyPut–Add a New Key and Element” on page 911
- “T02\_locate–Locate Key and Point Cursor to Its Element” on page 1155.



## TO2\_atKeyPut–Update the Specified Element Using a Key

This function searches the specified collection for the specified key and, if found, replaces the associated element value with the new element value.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long TO2_atKeyPut (const T02_PID_PTR  pid_ptr,
                    T02_ENV_PTR      env_ptr,
                    const void        *key,
                    const void        *data,
                    const long         *dataLength,
                    const long         *updateSeqCtr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the value that will be used as the key to locate the specific entry. The length of the key is assumed to be equal to the key length value specified when the collection was created.

#### data

The pointer to the element that will be stored in the specified entry.

#### dataLength

The pointer to an area where the length of the element will be stored.

#### updateSeqCtr

The pointer to an area where the update sequence counter from a T02\_atKey request has been stored. If this sequence counter value is not equal to the current update sequence counter value of the collection, the T02\_atKeyPut request will not be processed and an error will be returned.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_SEQCTR
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

## T02\_atKeyPut

### Programming Considerations

A commit scope will be created for the T02\_atKeyPut request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_atKeyPut request, the scope will be rolled back.

### Examples

The following example updates the entry in a given collection at the specified key.

```
#include <c$to2.h>                /* Needed for T02 API functions      */
#include <stdio.h>                /* APIs for standard I/O functions */
#include <stdlib.h>               /* APIs for standard library functions */

T02_PID      collect;            /* must set to existing collection's PID */
T02_BUF_PTR  elem_ptr;          /* will hold element from T02atKey      */
T02_ENV_PTR  env_ptr;           /* Pointer to the T02 environment       */
char         keya[] = "KeyA";    /* key to search for                    */
char         objectb[] = "ObjectB"; /* data                                */
long         objsizb;            /* size of data                         */
long         keylength;
:
:
/*****
/* First attempt to read item at keya to get sequence counter
*****/

keylength = sizeof(keya);

if ((elem_ptr = T02_atKey(&collect, env_ptr, keya, &keylength)) == T02_ERROR)
{
    printf("T02_atKey failed!\n");
    process_error(env_ptr);
    return;
}
/*****
/* Replace data at keya with objectb
*****/
objsizb = sizeof(objectb);
if (T02_atKeyPut(&collect, env_ptr, keya, objectb, &objsizb,
                &(elem_ptr->updateSeqNbr)) == T02_ERROR)
{
    printf("T02_atKeyPut failed!\n");
    process_error(env_ptr);
}
else
{
    printf("Data at %s has been changed to %s.\n", keya, objectb);
    printf("T02_atKeyPut is successful!\n");
}
```

### Related Information

- “T02\_atCursor–Return the Element Pointed to by the Cursor” on page 1119
- “T02\_atKey–Return the Specified Element by Key” on page 905
- “T02\_atKeyWithBuffer–Return the Specified Element” on page 909
- “T02\_atNewKeyPut–Add a New Key and Element” on page 911.

## TO2\_atKeyWithBuffer–Return the Specified Element

This function searches the specified collection for the specified key and, if found, returns the associated element value in the specified buffer. If the collection contains duplicate keys equal to the specified key, the first one found will be returned. The key is not returned if it is only partially matched.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_atKeyWithBuffer(const TO2_PID_PTR pid_ptr,
                                   TO2_ENV_PTR env_ptr,
                                   const void      *key,
                                   const long      *keylength,
                                   const long      *bufferLength,
                                   TO2_BUF_PTR buffer);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### key

The pointer to the value that will be used as the key to locate the specific entry.

#### keyLength

The pointer to the length of the key. The maximum length is 248 bytes. For key sorted set collections, the key length can be shorter than the defined key to allow a locate by partial key. The key comparison is always from the first character of the key for the specified length. For all other keyed collections, the key length must equal the maximum defined key length of the collection.

#### bufferLength

The pointer to the length of the specified buffer. The buffer should be large enough to hold the data to be returned plus 16 additional bytes for a header. The minimum buffer length is 20 bytes.

#### buffer

The pointer to the buffer where the element will be returned.

### Normal Return

The normal return value is the address of the same buffer that you specified when calling the function. The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

If the length of the data is greater than the length of the buffer, only the amount of data that will fit is placed in the buffer. The length is returned in the buffer and is the actual length of the data element. It is the responsibility of the caller to check the returned length against the length of the supplied buffer to determine if the entire element has been returned.

### Error Return

An error return is indicated by a NULL pointer. When a NULL pointer is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

## TO2\_atKeyWithBuffer

The following error codes are common for this function:

```
TO2_ERROR_BUFFER_SIZE
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example searches a keyed collection to determine if it contains an element with the specified key. If an element with that key is found, it is copied into the specified buffer.

```
#include <c$to2.h>                /* Needed for TO2 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

char          buf[50];
TO2_BUF_PTR   elem_ptr;
TO2_BUF_PTR   buf_ptr=&buf;
TO2_PID       collect;          /* will set to KeyBag PID */
TO2_ENV_PTR   env_ptr;          /* Pointer to the TO2 environment */
char          keya[] = "KeyA";  /* key to search for */
long          keylength;
long          bufL;
:
:
/*****
/* Test to see if collection contains entry for key in keya */
*****/

keylength = sizeof(keya);
bufL = sizeof(buf);

elem_ptr = TO2_atKeyWithBuffer(&collect, env_ptr, keya, &keylength,
                              &bufL, buf_ptr );
if (elem_ptr == NULL)
{
    printf("TO2_atKeyWithBuffer failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_atKeyWithBuffer is successful. Key found!\n");
}
```

## Related Information

- “TO2\_atCursor–Return the Element Pointed to by the Cursor” on page 1119
- “TO2\_atKey–Return the Specified Element by Key” on page 905
- “TO2\_atKeyPut–Update the Specified Element Using a Key” on page 907
- “TO2\_atNewKeyPut–Add a New Key and Element” on page 911.

## TO2\_atNewKeyPut—Add a New Key and Element

This function searches the specified collection for the specified key and, if not found, the key is added to the collection with the associated element value.

If the collection allows duplicate keys, the search is bypassed and the key and its associated element are added to the collection.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long TO2_atNewKeyPut (const T02_PID_PTR  pid_ptr,
                      T02_ENV_PTR  env_ptr,
                      const void      *key,
                      const void      *data,
                      const long      *dataLength);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the value that will be used as the key to locate the specific entry. The length of the key is assumed to be equal to the key length value specified when the collection was created.

#### data

The pointer to the element that will be stored in the specified entry.

#### dataLength

The pointer to an area that contains the length of the element that will be stored.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function.

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_UNIQUE
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

## TO2\_atNewKeyPut

### Programming Considerations

A commit scope will be created for the TO2\_atNewKeyPut request. If the request is successful, the scope will be committed. If an error occurs while processing the TO2\_atNewKeyPut request, the scope will be rolled back.

### Examples

The following example adds a new key and its associated data to the specified key bag collection.

```
#include <c$to2.h>                /* Needed for TO2 API functions */
#include <stdio.h>                /* APIs for std I/O functions */
#include <stdlib.h>              /* APIs for std library functions */

TO2_PID    collect;              /* must set to existing KeyBag PID */
TO2_ENV_PTR env_ptr;            /* Pointer to the TO2 environment */
char       keya[] = "KeyA";      /* key to search for */
char       objecta[] = "ObjectA"; /* data */
long       objsiza;             /* size of data */
:
:
/*****
/* Add key and its data to the collection.
*****/
objsiza = sizeof(objecta);
if (TO2_atNewKeyPut(&collect,
                  env_ptr,
                  keya,
                  objecta,
                  &objsiza) == TO2_ERROR)
{
    printf("TO2_atNewKeyPut failed!\n");
    process_error(env_ptr);
}
else
{
    printf("We have added key %s with data %s.\n", keya, objecta);
    printf("TO2_atNewKeyPut is successful!\n");
}
```

### Related Information

- “TO2\_atCursor–Return the Element Pointed to by the Cursor” on page 1119
- “TO2\_atKey–Return the Specified Element by Key” on page 905
- “TO2\_atKeyWithBuffer–Return the Specified Element” on page 909
- “TO2\_atKeyPut–Update the Specified Element Using a Key” on page 907.

## TO2\_atPut–Update the Specified Element

This function replaces the data of the specified entry with the new data that is supplied.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long TO2_atPut (const T02_PID_PTR pid_ptr,
                T02_ENV_PTR env_ptr,
                const long *index,
                const void *data,
                const long *dataLength,
                const long *updateSeqCtr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### index

The pointer to the entry number that will be updated with the specified data.

#### data

The pointer to the element that will be stored in the specified entry.

#### dataLength

The pointer to an area that contains the length of the element that will be stored.

#### updateSeqCtr

The pointer to an area where the update sequence counter from a T02\_at request has been stored. If this sequence counter value is not equal to the current update sequence counter value of the collection, the T02\_atPut request will not be processed and an error will be returned.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_INDEX
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_SEQCTR
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

## Programming Considerations

A commit scope will be created for the T02\_atPut request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_atPut request, the scope will be rolled back.

## Examples

The following example updates the third element of the specified collection with new data.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for std I/O functions */
#include <stdlib.h>              /* APIs for std library functions */

T02_PID      *collect;           /* must set to point to existing
                                collection's PID */
T02_BUF_PTR  elem_ptr;          /* will hold element from T02at */
T02_ENV_PTR  env_ptr;           /* Pointer to the T02 environment */
long         position = 3;      /* position to update */
const       newdata[] = "NewData"; /* data */
long         newdatasz;         /* data size */
:
:
/*****
/* First attempt to read entry at position to get sequence counter */
*****/

if ((elem_ptr = T02_atPut(&collect,
                        env_ptr,
                        &position)) == NULL)
{
    printf("T02_atPut failed.\n");
    process_error(env_ptr);
    return;
}
/*****
/* Replace data at that position with new data. */
*****/
newdatasz = sizeof(newdata);
if (T02_atPut(&collect, env_ptr, &position, newdata, &newdatasz,
             &(elem_ptr->updateSeqNbr)) == T02_ERROR)
{
    printf("T02_atPut failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_atPut is successful!\n");
}
```

## Related Information

- “T02\_at–Return the Specified Element by Index” on page 903
- “T02\_atCursor–Return the Element Pointed to by the Cursor” on page 1119
- “T02\_atWithBuffer–Retrieve an Element from a Collection” on page 922.



## TO2\_atRBA–Retrieve Data from a BLOB

This function can be called to read either a complete binary large object (BLOB) or only a part of a BLOB by specifying the starting byte position.

The number of bytes read will be the length specified (up to 4 MB (4 194 304)) or the size of the BLOB, whichever is smaller.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
void * TO2_atRBA (const T02_PID_PTR    pid_ptr,
                  T02_ENV_PTR        env_ptr,
                  const long          *relativeByteToRead,
                  long                *lengthToRead,
                  long                *updateSeqCtr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### relativeByteToRead

The pointer to a value that specifies the relative byte position in the collection to start the read. This position is 1-based.

#### lengthToRead

The pointer to a value that specifies the number of bytes that will be read. This field will be overlaid with the actual number of bytes read. The maximum length that can be read is 4 MB (4 194 304).

#### updateSeqCtr

The pointer to a field where the actual sequence counter value will be stored.

### Normal Return

The normal return is a pointer to a buffer containing the requested data. Once the caller has stopped using the returned data, it is the responsibility of the caller to free the data buffer. The **lengthToRead** parameter is updated with the actual number of bytes read.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_INDEX
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_REREAD
TO2_ERROR_SEQCTR
```

## TO2\_atRBA

TO2\_ERROR\_ZERO\_PID  
TO2\_ERROR\_NOT\_INIT

**Note:** The sequence counter field will be updated with the current sequence counter value from the BLOB whenever error codes TO2\_ERROR\_EODAD and TO2\_ERROR\_EMPTY are set in the environment.

## Programming Considerations

- It is the responsibility of the caller to free the returned buffer once the caller has stopped using the buffer.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example obtains a buffer with a copy of the data from a given BLOB starting at a specified relative byte position for a specified length.

```
#include <c$to2.h>          /* Needed for TO2 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

TO2_PID      blob;        /* Pointer to the TO2 environment */
TO2_ENV_PTR  env_ptr;     /* Pointer to the TO2 environment */
long         position=5;  /* We will try to get starting at 5 */
long         seq_ctr;     /* will hold update sequence count */
long         size=40;     /* number bytes to copy */
char         * buf_ptr;

:
:
/*****
/* Copy data from the BLOB starting at the 5th position for a
/* length of 40 bytes.
*****/

buf_ptr = TO2_atRBA(&blob,
                  env_ptr,
                  &position,
                  &size,
                  &seq_ctr);

if (buf_ptr == NULL)
{
    printf("TO2_atRBA failed !\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_atRBA successful!\n");
}
```

## Related Information

- “TO2\_atCursor–Return the Element Pointed to by the Cursor” on page 1119
- “TO2\_atRBAWithBuffer–Retrieve Data from a BLOB” on page 920
- “TO2\_atRBAPut–Store Data in a BLOB” on page 917
- “TO2\_getBLOB–Retrieve the Contents of a BLOB” on page 1045
- “TO2\_removeRBA–Remove an Area from a BLOB” on page 1084.

## TO2\_atRBAPut–Store Data in a BLOB

This function is used to either add data to a binary large object (BLOB), completely replace the contents of a current BLOB, or change a specific set of bytes in a BLOB. If the BLOB is larger than a single transmission buffer, multiple TO2\_atPut functions can be entered as each buffer is received to add the new data to the BLOB. The number of bytes stored will be the length specified (up to 4 MB (4 194 304)) or the size of the BLOB, whichever is smaller. To completely replace the data in an existing BLOB, call the TO2\_makeEmpty function before the first TO2\_atRBAPut is entered with the replacement data.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long TO2_atRBAPut (const T02_PID_PTR      pid_ptr,
                  T02_ENV_PTR            env_ptr,
                  const long              *startByteToPut,
                  const void              *data,
                  const long              *dataLength,
                  const long              *updateSeqCtr,
                  const T02_ATPUT_STATE  *moreData);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### startByteToPut

The pointer to a 1-based value that specifies the relative byte position in the collection to start storing the data.

#### data

The pointer to the data that will be stored in the collection.

#### dataLength

The pointer to an area that contains the number of bytes that will be stored.

The data written will be **all** or **none** because no partial writes are allowed. The maximum length that can be read is 4 MB (4 194 304).

#### updateSeqCtr

The pointer to an area where the update sequence counter from a TO2\_atRBA request has been stored. If this sequence counter value is not equal to the current update sequence counter value of the collection, the TO2\_atRBAPut request will not be processed.

#### moreData

A pointer to a character that signals if there is more data that will be placed in the BLOB, or if this is the last or only TO2\_atRBAPut request that will be entered for the BLOB.

#### TO2\_ATPUT\_ONLY

This is the only TO2\_atRBAPut function call. The BLOB is completed and can be closed out.

#### TO2\_ATPUT\_FIRST

This is the first function call and there is more data that will be placed in the BLOB using TO2\_atRBAPut. The BLOB will not be closed out.

## TO2\_atRBAPut

### TO2\_ATPUT\_MORE

This is not the first call and there is more data that will be placed in the BLOB using TO2\_atRBAPut. The BLOB will not be closed out.

### TO2\_ATPUT\_LAST

Last TO2\_atRBAPut. The BLOB is completed and can be closed out.

## Normal Return

The normal return is a positive value.

## Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ACCESS_MISMATCH
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_INDEX
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_SEQCTR
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- A commit scope will be created for the TO2\_atRBAPut request. If the request is successful, the scope will be committed. If an error occurs while processing the TO2\_atRBAPut request, the scope will be rolled back.
- If the wrong moreData indicators are used, such as not using TO2\_ATPUT\_LAST after TO2\_ATPUT\_FIRST, results cannot be predicted and system errors may occur.
- The update sequence counter is updated after every TO2\_atRBAPut request, regardless of the value of the moreData parameter. The application must either keep the sequence counter after the first TO2\_atRBA request and increment it by one after each TO2\_atRBAPut request, or the TO2\_atRBA function must be called before each TO2\_atRBAPut request.

## Examples

The following example resets specific bytes of a specified BLOB.

```
#include <c$to2.h>           /* Needed for TO2 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */
#include <stdlib.h>          /* APIs for standard library functions */

TO2_PID      blob;
TO2_ENV_PTR  env_ptr;      /* Pointer to the TO2 environment */
long         position=5;    /* We will try to get starting at 5 */
long         seq_ctr;      /* will hold update sequence count */
long         size=8;       /* number bytes to read/write */
mode         long;
char         *buff_ptr;
char         newdata[] = "NewData"; /* new data */
TO2_ATPUT_STATE  moreData = TO2_ATPUT_ONLY;
```

```

:
/*****
/* Look at the BLOB to set the contents of sequence counter for
/* to use for subsequent update.
*****/

buff_ptr = TO2_atRBA(&blob,
                    env_ptr,
                    &position,
                    &size,
                    &seq_ctr);
if (buff_ptr == NULL)
{
    printf("TO2_atRBA failed!\n");
    process_error(env_ptr);
    exit(0);
}
:
free(buff_ptr);
/*****
/* Now update BLOB to change data starting at position 5 to new
/* data...
*****/

size = sizeof(newdata);
if (TO2_atRBAPut(&blob,
                env_ptr,
                &position,
                newdata,
                &size,
                &seq_ctr,
                &moreData) == TO2_ERROR)
{
    printf("TO2_atRBAPut failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_atRBAPut successful!\n");
}

```

## Related Information

- “TO2\_atCursor—Return the Element Pointed to by the Cursor” on page 1119
- “TO2\_atRBAWithBuffer—Retrieve Data from a BLOB” on page 920
- “TO2\_removeRBA—Remove an Area from a BLOB” on page 1084
- “TO2\_replaceBLOB—Replace the Contents of a BLOB with New Data” on page 1092.

## TO2\_atRBAWithBuffer–Retrieve Data from a BLOB

This function can be called to read either a complete binary large object (BLOB), or only part of a BLOB by specifying the starting byte position.

The data that is read is placed in the buffer specified by the caller. The number of bytes read will be the length specified, the maximum buffer length of 32 KB (8 × 4096), or the size of the BLOB, whichever is smaller.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
void * TO2_atRBAWithBuffer (const T02_PID_PTR pid_ptr,
                             T02_ENV_PTR env_ptr,
                             const long      *relativeByteToRead,
                             long            *bufferLength,
                             long            *updateSeqCtr,
                             void            *buffer);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### relativeByteToRead

The pointer to a value that specifies the relative byte position in the collection to start the read.

#### bufferLength

The pointer to a value that specifies the length of the specified buffer. This field will be overlaid with the actual number of bytes read. If the returned length is less than the size of the buffer, the length field is updated with the actual length returned. The minimum buffer length is 20 bytes and the maximum is 32 768.

#### updateSeqCtr

The pointer to a field where the actual sequence counter value will be stored.

#### buffer

The pointer to a buffer where the read data is returned.

### Normal Return

The normal return is a pointer to the buffer (array of character) supplied by the caller. This buffer now contains the requested data.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_BUFFER_SIZE
TO2_ERROR_DATA_LGH
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_INDEX
```

```

TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_SEQCTR
TO2_ERROR_ZERO_PID

```

**Note:** The sequence counter field will be updated with the current sequence counter value from the BLOB whenever error codes TO2\_ERROR\_EODAD and TO2\_ERROR\_EMPTY are set in the environment.

## Programming Considerations

- This function does not use TPF transaction services on behalf of the caller.
- Processing continues if a duplicate element is added to a unique collection. All other errors cause no change to the target collection.

## Examples

The following example copies data into a specified buffer from a given BLOB starting at a specified relative byte position for a specified length.

```

#include <c$to2.h>          /* Needed for TO2 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

TO2_PID          blob;
TO2_ENV_PTR      env_ptr; /* Pointer to the TO2 environment */
long             position=5; /* We will try to get starting at 5 */
long             seq_ctr;   /* will hold update sequence count */
char             buff[40];

/* use buffer */

char             * buff_ptr;
long             buffL;    /* length of user buffer */
:
:
/*****
/* Copy data from the BLOB starting at the 5th position for up
/* to 40 bytes, that is the size of the user buffer.
*****/

buffL = sizeof(buff);
buff_ptr = TO2_atRBAWithBuffer(&blob,
                               env_ptr,
                               &position,
                               &buffL,
                               &seq_ctr,
                               buff);

if (buff_ptr == NULL)
{
    printf("TO2_atRBAWithBuffer failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_atRBAWithBuffer successful!\n");
}

```

## Related Information

- “TO2\_atCursor–Return the Element Pointed to by the Cursor” on page 1119
- “TO2\_atRBA–Retrieve Data from a BLOB” on page 915
- “TO2\_atRBAPut–Store Data in a BLOB” on page 917
- “TO2\_getBLOB–Retrieve the Contents of a BLOB” on page 1045
- “TO2\_removeRBA–Remove an Area from a BLOB” on page 1084.

## TO2\_atWithBuffer–Retrieve an Element from a Collection

This function returns the data at the specified index in the specified collection in the supplied buffer. If the collection is an array and the position is marked as empty, this method returns a data length of 0 and an update sequence counter of 0.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_atWithBuffer (const TO2_PID_PTR  pid_ptr,
                                TO2_ENV_PTR    env_ptr,
                                const long      *index,
                                const long      *bufferLength,
                                TO2_BUF_PTR    buffer);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### index

The pointer to the index of the entry that has contents that will be returned.

#### bufferLength

The pointer to a field that contains the length of the supplied buffer. The buffer should be large enough to hold the data to be returned plus 16 additional bytes for a header. The minimum buffer length is 20 bytes.

#### buffer

The pointer to the supplied buffer.

### Normal Return

The normal return value is the address of the same buffer that the you specified when calling the function. The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

If the length of the data is greater than the length of the buffer, only the amount of data that will fit is placed in the buffer. The length is returned in the buffer and is the actual length of the data element. It is the responsibility of the caller to check the returned length against the length of the supplied buffer to determine if the entire element has been returned.

### Error Return

An error return is indicated by a NULL pointer. When a NULL pointer is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_BUFFER_SIZE
TO2_ERROR_DATA_LGH
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_INDEX
TO2_ERROR_METHOD
```



```

TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID

```

## Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example copies the third element of a specified collection into the specified buffer.

```

#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

char          [100];
long          my_bufL;
T02_PID       collect;
T02_ENV_PTR   env_ptr;          /* Pointer to the T02 environment */
long          position=3;       /* We will try to get third element */
T02_ERROR_CODE err_code;
:
/*****
/* Copy the element at the specified position.
*****/
my_bufL = sizeof(my_buf);
if (T02_atWithBuffer == NULL (&collect,
                             env_ptr,
                             &position,
                             &my_bufL,
                             my_buf));

if (my_buf == NULL)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_ERROR_EODAD)
    {
        printf("T02_atWithBuffer failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_atWithBuffer successful!\n");
}

```

## Related Information

- “TO2\_at–Return the Specified Element by Index” on page 903
- “TO2\_atCursor–Return the Element Pointed to by the Cursor” on page 1119
- “TO2\_atPut–Update the Specified Element” on page 913
- “TO2\_removeIndex–Remove Element from the Index” on page 1078.

## TO2\_capture—Capture a Collection to an External Device

This function causes the specified collection to be written to an external device along with all associated data required to restore the collection to the database.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_capture (const TO2_PID_PTR  pid_ptr,
                          TO2_ENV_PTR      env_ptr,
                          const void       *externalToken,
                          const void       *data,
                          const long       *dataLength,
                          const long       *mountTimeOut);
```

#### **pid\_ptr**

The pointer to the persistent identifier (PID) assigned to the collection that will be captured.

#### **env\_ptr**

The pointer to the environment as returned by the TO2\_createEnv function.

#### **externalToken**

The token returned by the TPFxd\_archiveStart function call.

#### **data**

The user data to write to the external device.

#### **dataLength**

The pointer to an area that contains the length of the user data.

#### **mountTimeOut**

The pointer to an area that contains the number of seconds to wait for the mount of the external device.

### Normal Return

For a normal return, the TO2\_capture function will return a buffer item that contains the required positioning information for a TO2\_restore request. The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, the TO2\_getErrorCode function can be used to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_METHOD
TO2_ERROR_NO_XTERNAL_DEFINED
TO2_ERROR_NO_XTERNAL_DEVICES
TO2_ERROR_NOT_INIT
TO2_ERROR_PERMANENT_XTERNAL
TO2_ERROR_PID
TO2_ERROR_TIMEOUT
TO2_ERROR_ZERO_PID
TO2xd_ERROR_tapeBlocked
```

## Programming Considerations

- Collections cannot be captured or restored using blocked tapes.
- You must enter explicit TPFxd\_archiveStart and TPFxd\_archiveEnd function calls before and after using this function.
- The information in the returned buffer is needed on subsequent T02\_restore... functions.
- The specified user data is written along with the collection and is returned on the T02\_restore request.
- The T02\_capture function will return positioning information that is required for the T02\_restore function to restore the collection.
- If the collection also has assigned properties, the properties will also be captured as part of the T02\_capture request.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example causes the specified collection to be written to an external device.

```
#include <c$to2.h>
#include <c$tpxd.h>

T02_PID      capturePID;
T02_ENV_PTR  env_ptr=NULL;
T02_BUF_PTR  buffer_ptr;
TPFxd_extToken *token_ptr;
char         data[80];
long         dataLength=sizeof(data);
long         return_code;
long         timeOut=5*60;    /* time out mount in 5 minutes */
:
:
/*****
/* We need to activate the archiving external device support by */
/* obtaining a token. Upon a successful return, the token will */
/* be used on every T02_capture call for this archive run.      */
*****/
if ((return_code = TPFxd_archiveStart(&token_ptr,
                                     WT,
                                     ASAVAILABLE)) != TPFxd_SUCCESS)
{
    printf("TPFxd_archiveStart failed!\n");
    process_TPFerror();
    return();
}

if ((buffer_ptr = T02_capture(&capturePID,
                             env_ptr,
                             token_ptr,
                             data,
                             &dataLength,
                             &timeOut)) == NULL);
{
    printf("T02_capture failed!\n");
    process_error();
}
else
    printf("T02_capture successful!\n");
    TPFxd_archiveEnd()
```

## TO2\_capture

### Related Information

- “TPFxd\_archiveEnd—Inform the Archive Facility That the Request Has Ended” on page 1360
- “TPFxd\_archiveStart—Start the Archive Support Facility” on page 1361
- “TO2\_restore—Restore a Previously Captured Collection” on page 1094
- “TO2\_restoreWithOptions—Restore a Collection Using the Specified Options” on page 1098
- “TO2\_restoreAsTemp—Restore a Collection as a Temporary Collection” on page 1096.

## T02\_copyCollection—Make a Persistent Copy of the Collection

This function creates a copy of the specified source collection, including the following:

- Access mode
- Associated recoup index
- Dirty-read (DR) protect mode
- Key paths
- Properties
- Read-only attribute
- User class ID.

### Format

```
#include <c$to2.h>
long T02_copyCollection (const T02_PID_PTR  pid_ptr,
                           T02_ENV_PTR    env_ptr,
                           T02_PID_PTR    rpid_ptr);
```

#### **pid\_ptr**

The pointer to the PID assigned to the collection that will be copied.

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **rpidd\_ptr**

The pointer to where the PID for the copy of the collection will be stored.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_NOT_INIT
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

- The copy of the collection is persistent and will not be deleted automatically. To delete it, use the T02\_deleteCollection function.
- A commit scope will be created for the T02\_copyCollection request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_copyCollection request, the scope will be rolled back.

### Examples

The following example creates a persistent copy of an existing collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */
```

## T02\_copyCollection

```
T02_ENV_PTR    env_ptr;      /* Pointer to the T02 environment */
T02_PID        sourcePID;    /* must create collection and save
                             its PID here prior to copy... */
T02_PID        copyPID;      /* will obtain PID for copied
                             collection below... */
:
if (T02_copyCollection(&sourcePID,
                      env_ptr,
                      &copyPID) == T02_ERROR)
{
    printf("T02_copyCollection failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_copyCollection successful!\n");
}
```

## Related Information

- “TO2\_addAllFrom–Add All from Source Collection” on page 886
- “TO2\_copyCollectionTemp–Make a Temporary Copy of the Collection” on page 929
- “TO2\_copyCollectionWithOptions–Make a Copy Using Options” on page 931
- “TO2\_deleteCollection–Delete a Collection” on page 1036
- “TO2\_migrateCollection–Migrate a Collection” on page 1318.

See *TPF Application Programming* for more information about commit scope.

## T02\_copyCollectionTemp—Make a Temporary Copy of the Collection

This function creates a temporary copy of the specified collection and returns its persistent identifier (PID). The copy is created using the default data definition for a temporary collection of its type. The function does **not** copy the properties of the source collection.

### Format

```
#include <c$to2.h>
long T02_copyCollectionTemp (const T02_PID_PTR  pid_ptr,
                               T02_ENV_PTR    env_ptr,
                               T02_PID_PTR    rpid_ptr);
```

#### pid\_ptr

The pointer to the PID assigned to the collection that will be copied.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### rpidd\_ptr

The pointer to where the PID for the copy of the collection will be returned.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_NOT_INIT
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the entry control block (ECB) exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function cannot be used to copy a collection that has alternate key paths defined.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of a temporary collection after the ECB that created it exits will cause unexpected results.

### Examples

The following example creates a temporary copy of an existing collection.

```
#include <c$to2.h>          /* Needed for T02 API functions    */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* Pointer to the T02 environment */
```

## TO2\_copyCollectionTemp

```
TO2_PID      sourcePID;    /* must create collection and save
                           its PID here prior to copy... */
TO2_PID      copyPID;      /* will obtain PID for copied
                           collection below... */
:
if (TO2_copyCollectionTemp(&sourcePID,
                           env_ptr,
                           &copyPID) == TO2_ERROR)
{
    printf("TO2_copyCollectionTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_copyCollectionTemp successful!\n");
}
:
TO2_deleteCollection(&copyPID, env_ptr);
```

## Related Information

- “TO2\_addAllFrom—Add All from Source Collection” on page 886
- “TO2\_copyCollection—Make a Persistent Copy of the Collection” on page 927
- “TO2\_copyCollectionWithOptions—Make a Copy Using Options” on page 931
- “TO2\_deleteCollection—Delete a Collection” on page 1036.



## T02\_copyCollectionWithOptions–Make a Copy Using Options

This function creates a copy of the specified collection using the specified data definition and returns its persistent identifier (PID) as a parameter. This function does **not** copy the properties of the source collection.

### Format

```
#include <c$to2.h>
long T02_copyCollectionWithOptions (const T02_PID_PTR    sourceCollect,
                                     T02_ENV_PTR        env_ptr,
                                     T02_PID_PTR        copyCollect,
                                     const T02_OPTION_PTR optionListPtr);
```

#### **sourceCollect**

The pointer to the PID assigned to the collection that will be copied.

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **copyCollect**

The pointer to where the PID for the copy of the collection will be stored.

#### **optionListPtr**

The pointer to a returned options list from a T02\_createOptionList function call or NULL if no options are required.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DDNAME
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- A commit scope will be created for the T02\_copyCollectionWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_copyCollectionWithOptions request, the scope will be rolled back.
- This function cannot be used to copy a collection that has alternate key paths defined.

### Examples

The following example creates a persistent copy of an existing collection using a specified data definition.

```
#include <c$to2.h>                                     /* Needed for T02 API functions */
T02_ENV_PTR    env_ptr;                                /* Pointer to the T02 environment */
```

## T02\_copyCollectionWithOptions

```
T02_PID      sourceCollect;    /* must create collection and save
                                its PID here before copy... */
T02_PID      copyCollect;      /* will obtain PID for copied
                                collection below... */

T02_OPTION_PTR optionListPtr;
u_char dd_name[] = "NAME_OF_USER_DD_DEFINITION";
:
optionListPtr=T02_createOptionList(env_ptr,
                                T02_OPTION_LIST_CREATE, /* create options */
                                T02_CREATE_DD,          /* use provided dd name*/
                                T02_OPTION_LIST_END,     /* end of options */
                                dd_name);               /* address of ddname */

if (T02_copyCollectionWithOptions(&sourceCollect,
                                env_ptr,
                                &copyCollect,
                                optionListPtr) == T02_ERROR)
{
    printf("T02_copyCollectionWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_copyCollectionWithOptions successful!\n");
}
```

## Related Information

- “TO2\_addAllFrom—Add All from Source Collection” on page 886
- “TO2\_copyCollection—Make a Persistent Copy of the Collection” on page 927
- “TO2\_createOptionList—Create a TPF Collection Support Option List” on page 990.

## T02\_createArray—Create a Persistent Array Collection

This function creates an empty persistent array collection using the default data definition and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createArray (      T02_PID_PTR  pid_ptr,
                           T02_ENV_PTR  env_ptr,
                           const long    *entryLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 4000 bytes. For an array, all entries must be the same length.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_DATA_LGH
```

## Programming Considerations

A commit scope will be created for the T02\_createArray request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createArray request, the scope will be rolled back.

## Examples

The following example creates a persistent array collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;        /* PTR to the T02 environment */
T02_PID_PTR  collect;        /* will hold collection's PID */
long         entryLength=100; /* set to entry length */
:
:
if (T02_createArray(&collect,
                  env_ptr,
                  &entryLength) == T02_ERROR)
{
    printf("T02_createArray failed!\n");
    process_error(env_ptr);
}
```

## TO2\_createArray

```
else
{
    printf("TO2_createArray successful!\n");
}
```

## Related Information

- “TO2\_createArrayTemp—Create a Temporary Array Collection” on page 935
- “TO2\_createArrayWithOptions—Create an Empty Array Collection” on page 937.

## T02\_createArrayTemp—Create a Temporary Array Collection

This function creates a temporary array collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createArrayTemp (      T02_PID_PTR   pid_ptr,
                                T02_ENV_PTR   env_ptr,
                                const long    *entryLength);
```

#### pid\_ptr

The pointer to the location where the temporary PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 4000 bytes. For an array, all entries must be the same length.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

### Examples

The following example creates a temporary array collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_PID    collect;              /* will hold collection's PID */
long       entryLength=100;      /* set to entry length */
:
:
if (T02_createArrayTemp(&collect,
                        env_ptr,
                        &entryLength) == T02_ERROR)
```

## TO2\_createArrayTemp

```
{
    printf("TO2_createArrayTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_createArrayTemp successful!\n");
}
:
TO2_deleteCollection(&collect, env_ptr);
```

## Related Information

- “TO2\_createArray—Create a Persistent Array Collection” on page 933
- “TO2\_createArrayWithOptions—Create an Empty Array Collection” on page 937
- “TO2\_deleteCollection—Delete a Collection” on page 1036.

## T02\_createArrayWithOptions—Create an Empty Array Collection

This function creates an empty array collection using the specified options and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createArrayWithOptions (      T02_PID_PTR      pid_ptr,
                                       T02_ENV_PTR      env_ptr,
                                       const T02_OPTION_PTR optionListPtr,
                                       const long        *entryLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList function call or NULL if no options are required.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 4000 bytes. For an array, all entries must be the same length.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_BUFFER_SIZE
TO2_ERROR_DATA_LGH
TO2_ERROR_DDNAME
TO2_ERROR_ENV
```

### Programming Considerations

- A commit scope will be created for the T02\_createArrayWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createArrayWithOptions request, the scope will be rolled back.
- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

### Examples

The following example creates a persistent array collection using specified options.

```
#include <c$to2.h>                                /* Needed for T02 API functions */
#include <stdio.h>                                /* APIs for standard I/O funcs */
```

## TO2\_createArrayWithOptions

```
TO2_ENV_PTR    env_ptr;           /* Pointer to the T02 envrnmnt */
TO2_PID        collect;          /* will hold collection's PID */
char           dd_name[] = "NAME_OF_USER_DD_DEFINITION" ;
long           entryLength=300;   /* set to max entry length */
TO2_OPTION_PTR optionListPtr;
char           recoupname[] = "HOSP0001";

/* invoke T02 to create opt list */
:
optionListPtr T02_createOptionList(env_ptr,
                                   T02_OPTION_LIST_CREATE, /* create options */
                                   T02_CREATE_SHADOW,      /* use shadowing */
                                   T02_CREATE_DD,           /* use provided ddname*/
                                   T02_CREATE_RECOUP,       /* use provided recoup*/
                                   T02_OPTION_LIST_END,     /* end of options */
                                   dd_name,                 /* address of ddname */
                                   recoupname);             /* addr of recoup key */

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createArrayWithOptions(&collect,
                               env_ptr,
                               optionListPtr,
                               &entryLength) == T02_ERROR)
{
    printf("T02_createArrayWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createArrayWithOptions successful!\n");
}
```

## Related Information

- “TO2\_createArray—Create a Persistent Array Collection” on page 933
- “TO2\_createArrayTemp—Create a Temporary Array Collection” on page 935
- “TO2\_createOptionList—Create a TPF Collection Support Option List” on page 990.



## TO2\_createBag—Create a Persistent Bag Collection

This function creates an empty persistent bag collection using the default data definition and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long TO2_createBag (      T02_PID_PTR   pid_ptr,
                          T02_ENV_PTR   env_ptr,
                          const long    *entryLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 248 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
```

## Programming Considerations

A commit scope will be created for the T02\_createBag request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createBag request, the scope will be rolled back.

## Examples

The following example creates a persistent bag collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;      /* PTR to the T02 environment */
T02_PID       collect;      /* will hold collection's PID */
long          entryLength=100; /* set to entry length */
:
:
if (T02_createBag(&collect,
                 env_ptr,
                 &entryLength) == T02_ERROR)
{
    printf("T02_createBag failed!\n");
    process_error(env_ptr);
}
```

## TO2\_createBag

```
else
{
    printf("TO2_createBag successful!\n");
}
```

## Related Information

- “TO2\_createBagTemp—Create a Temporary Bag Collection” on page 941
- “TO2\_createBagWithOptions—Create an Empty Bag Collection” on page 943.

## T02\_createBagTemp—Create a Temporary Bag Collection

This function creates a temporary bag collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createBagTemp (      T02_PID_PTR   pid_ptr,
                             T02_ENV_PTR   env_ptr,
                             const long    *entryLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 248 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

### Examples

The following example creates a temporary bag collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;      /* PTR to the T02 environment */
T02_PID       collect;      /* will hold collection's PID */
long          entryLength=100; /* set to entry length */
:
:
if (T02_createBagTemp(&collect,
                    env_ptr,
```

## T02\_createBagTemp

```
                                &entryLength) == T02_ERROR)
{
    printf("T02_createBagTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createBagTemp successful!\n");
}
:
T02_deleteCollection(&collect, env_ptr);
```

## Related Information

- “T02\_createBag—Create a Persistent Bag Collection” on page 939
- “T02\_createBagWithOptions—Create an Empty Bag Collection” on page 943
- “T02\_deleteCollection—Delete a Collection” on page 1036.

## T02\_createBagWithOptions—Create an Empty Bag Collection

This function creates an empty bag collection using the specified options and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createBagWithOptions (      T02_PID_PTR    pid_ptr,
                                     T02_ENV_PTR    env_ptr,
                                     const T02_OPTION_PTR optionListPtr);
                                     const long      *entryLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList function call or NULL if no options are required.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 248 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_DDNAME
T02_ERROR_ENV
```

### Programming Considerations

- A commit scope will be created for the T02\_createBagWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createBagWithOptions request, the scope will be rolled back.
- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

### Examples

The following example creates a persistent bag collection using specified options.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
T02_PID        collect;         /* will hold collection's PID */
```

## TO2\_createBagWithOptions

```
u_char      dd_name[] = "NAME_OF_USER_DD_DEFINITION" ;
long        entryLength=200; /* set to max entry length */
T02_OPTION_PTR optionListPtr;
char        recoupname[] = "HOSP0001";

/* invoke T02 to create option list */
:
optionListPtr = T02_createOptionList(env_ptr,
                                     T02_OPTION_LIST_CREATE, /* create options */
                                     T02_CREATE_SHADOW,      /* use shadowing */
                                     T02_CREATE_DD,           /* use provided dd name */
                                     T02_CREATE_RECOUP,        /* use provided recoup */
                                     T02_OPTION_LIST_END,      /* end of options */
                                     dd_name,                 /* address of ddname */
                                     recoupname);              /* addr of recoup key */

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createBagWithOptions(&collect,
                             env_ptr,
                             optionListPtr,
                             &entryLength) == T02_ERROR)
{
    printf("T02_createBagWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createBagWithOptions successful!\n");
}
```

## Related Information

- “TO2\_createBag—Create a Persistent Bag Collection” on page 939
- “TO2\_createBagTemp—Create a Temporary Bag Collection” on page 941
- “TO2\_createOptionList—Create a TPF Collection Support Option List” on page 990.

## T02\_createBLOB—Create a Persistent BLOB Collection

This function creates an empty persistent binary large object (BLOB) collection using the default data definition (DD) and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createBLOB (T02_PID_PTR pid_ptr,
                    T02_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
```

## Programming Considerations

A commit scope will be created for the T02\_createBLOB request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createBLOB request, the scope will be rolled back.

## Examples

The following example creates a persistent BLOB collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;       /* PTR to the T02 environment */
T02_PID_PTR collect;       /* will hold collection's PID */
:
:
if (T02_createBLOB(&collect,
                  env_ptr) == T02_ERROR)
{
    printf("T02_createBLOB failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createBLOB successful!\n");
}
```

## TO2\_createBLOB

### Related Information

- “TO2\_createBLOBTemp—Create a Temporary BLOB Collection” on page 947
- “TO2\_createBLOBWithOptions—Create an Empty BLOB Collection” on page 949
- “TO2\_writeNewBLOB—Create a New BLOB and Add the Passed Data” on page 1111.



## T02\_createBLOBTemp—Create a Temporary BLOB Collection

This function creates a temporary binary large object (BLOB) collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createBLOBTemp (T02_PID_PTR   pid_ptr,
                        T02_ENV_PTR   env_ptr);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
```

## Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

## Examples

The following example creates a temporary BLOB collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;          /* PTR to the T02 environment */
T02_PID_PTR   collect;          /* will hold collection's PID */
:
:
if (T02_createBLOBTemp(&collect,
                      env_ptr) == T02_ERROR)
{
    printf("T02_createBLOBTemp failed!\n");
    process_error(env_ptr);
}
else
{
```

## TO2\_createBLOBTemp

```
    printf("TO2_createBLOBTemp successful!\n");  
}  
:  
TO2_deleteCollection(&collect, env_ptr);
```

## Related Information

- “TO2\_createBLOB—Create a Persistent BLOB Collection” on page 945
- “TO2\_createBLOBWithOptions—Create an Empty BLOB Collection” on page 949
- “TO2\_deleteCollection—Delete a Collection” on page 1036
- “TO2\_writeNewBLOB—Create a New BLOB and Add the Passed Data” on page 1111.

## TO2\_createBLOBWithOptions—Create an Empty BLOB Collection

This function creates an empty binary large object (BLOB) collection using the specified options and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long TO2_createBLOBWithOptions (      T02_PID_PTR    pid_ptr,
                                      T02_ENV_PTR    env_ptr,
                                      const T02_OPTION_PTR optionListPtr);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### optionListPtr

The pointer to a returned options list from a TO2\_createOptionList function call or NULL if no options are required.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_DDNAME
TO2_ERROR_ENV
```

## Programming Considerations

- A commit scope will be created for the TO2\_createBLOBWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the TO2\_createBLOBWithOptions request, the scope will be rolled back.
- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

## Examples

The following example creates a persistent BLOB collection using the specified options.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
T02_PID        collect;         /* will hold collection's PID */
u_char        dd_name[] = "NAME_OF_USER_DD_DEFINITION";
T02_OPTION_PTR optionListPtr;
char          recoupname[] = "HOSP0001";
```

## TO2\_createBLOBWithOptions

```

                                /* invoke T02 to create option list */
                                :
optionListPtr T02_createOptionList(env_ptr,
                                T02_OPTION_LIST_CREATE, /* create options */
                                T02_CREATE_SHADOW,      /* use shadowing */
                                T02_CREATE_DD,           /* use provided ddnm */
                                T02_CREATE_RECOUP,       /* use provided recp */
                                T02_OPTION_LIST_END,     /* end of options */
                                dd_name,                 /* address of ddname */
                                recoupname);             /* addr of recoup key*/

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createBLOBWithOptions(&collect,
                              env_ptr,
                              optionListPtr) == T02_ERROR)
{
    printf("T02_createBLOBWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createBLOBWithOptions successful!\n");
}
}
```

## Related Information

- “TO2\_createBLOB—Create a Persistent BLOB Collection” on page 945
- “TO2\_createBLOBTemp—Create a Temporary BLOB Collection” on page 947
- “TO2\_createOptionList—Create a TPF Collection Support Option List” on page 990.

---

## TO2\_createDictionary—Create a Persistent Dictionary Collection

This function has been renamed to TO2\_createKeySortedSet. For more information, see “TO2\_createKeySortedSet—Create an Empty Persistent Key Sorted Set Collection” on page 978.

## **T02\_createDictionaryTemp—Create a Temporary Dictionary**

This function has been renamed to T02\_createKeySortedSetTemp. For more information, see “T02\_createKeySortedSetTemp—Create a Temporary Key Sorted Set” on page 980.

---

## **T02\_createDictionaryWithOptions—Create an Empty Dictionary Collection**

This function has been renamed to T02\_createKeySortedSetWithOptions. For more information, see “T02\_createKeySortedSetWithOptions—Create an Empty Key Sorted Set Collection” on page 982.

## T02\_createKeyBag—Create a Persistent Key Bag Collection

This function creates an empty persistent key bag collection using the default data definition (DD) and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createKeyBag (      T02_PID_PTR    pid_ptr,
                             T02_ENV_PTR    env_ptr,
                             const long     *entryLength,
                             const long     *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 1000 bytes.

#### keyLength

The pointer to the length of the key. The maximum length is 248 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

A commit scope will be created for the T02\_createKeyBag request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createKeyBag request, the scope will be rolled back.

### Examples

The following example creates a persistent key bag collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* PTR to the T02 environment */
T02_PID        collect;          /* will hold collection's PID */
long           entryLength=100;  /* set to entry length */
long           keyLength=8;      /* set to key length */
:
:
if (T02_createKeyBag(&collect,
                    env_ptr,
                    &entryLength,
                    &keyLength) == T02_ERROR)
```



```
{  
    printf("T02_createKeyBag failed!\n");  
    process_error(env_ptr);  
}  
else  
{  
    printf("T02_createKeyBag successful!\n");  
}
```

## Related Information

- “T02\_createKeyBagWithOptions—Create an Empty Key Bag Collection with Options” on page 958
- “T02\_createKeyBagTemp—Create a Temporary Key Bag Collection” on page 956.

## T02\_createKeyBagTemp—Create a Temporary Key Bag Collection

This function creates a temporary key bag collection and assigns it a persistent identifier (PID). A temporary key bag collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createKeyBagTemp (      T02_PID_PTR   pid_ptr,
                                T02_ENV_PTR   env_ptr,
                                const long    *entryLength,
                                const long    *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length depends on the collection. For a key bag collection, the length is 1000 bytes.

#### keyLength

The pointer to the length of the key. The maximum length is 248 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

### Examples

The following example creates a temporary key bag collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;     /* PTR to the T02 environment */
```

## TO2\_createKeyBagTemp

```
TO2_PID      collect;      /* will hold collection's PID      */
long         entryLength=100; /* set to entry length      */
long         keylength=8;    /* set to key length        */
:
:
if (TO2_createKeyBagTemp(&collect,
                        env_ptr,
                        &entryLength,
                        &keyLength) == TO2_ERROR)
{
    printf("TO2_createKeyBagTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_createKeyBagTemp successful!\n");
}
:
TO2_deleteCollection(&collect, env_ptr);
```

## Related Information

- “TO2\_createKeyBag—Create a Persistent Key Bag Collection” on page 954
- “TO2\_createKeyBagWithOptions—Create an Empty Key Bag Collection with Options” on page 958
- “TO2\_deleteCollection—Delete a Collection” on page 1036.

## T02\_createKeyBagWithOptions—Create an Empty Key Bag Collection with Options

This function creates an empty key bag collection using the specified options and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createKeyBagWithOptions (      T02_PID_PTR    pid_ptr,
                                         T02_ENV_PTR    env_ptr,
                                         const T02_OPTION_PTR optionListPtr);
                                         const long      *entryLength,
                                         const long      *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList call or NULL if no options are required.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 1000 bytes.

#### keyLength

The pointer to the length of the key. The maximum length is 248 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_DDNAME
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

- A commit scope will be created for the T02\_createKeyBagWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createKeyBagWithOptions request, the scope will be rolled back.
- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

## Examples

The following example creates a persistent key bag collection using the specified options.

```
#include <c$to2.h>          /* Needed for T02 API functions      */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* Pointer to the T02 environment */
T02_PID        collect;    /* will hold collection's PID      */
u_char         dd_name[] = "NAME_OF_USER_DD_DEFINITION"
long           entryLength=300; /* set to max entry length      */
long           keyLength=8;   /* set to key length             */
T02_OPTION_PTR optionListPtr;
char           recoupname[] = "HOSP0001";

                                /* invoke T02 to create option list */
                                :
                                :
optionListPtr = T02_createOptionList(env_ptr,
                                     T02_OPTION_LIST_CREATE, /* create options */
                                     T02_CREATE_SHADOW,      /* use shadowing  */
                                     T02_CREATE_DD,           /* use provided dd name*/
                                     T02_CREATE_RECOUP,        /* use provided recoup */
                                     T02_OPTION_LIST_END,      /* end of options   */
                                     dd_name,                 /* address of ddname */
                                     recoupname);              /* addr of recoup key */

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createKeyBagWithOptions(&collect,
                                env_ptr,
                                optionListPtr,
                                &entryLength,
                                &keylength) == T02_ERROR)
{
    printf("T02_createKeyBagWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createKeyBagWithOptions successful!\n");
}
```

## Related Information

- “T02\_createKeyBag—Create a Persistent Key Bag Collection” on page 954
- “T02\_createKeyBagTemp—Create a Temporary Key Bag Collection” on page 956
- “T02\_createOptionList—Create a TPF Collection Support Option List” on page 990.

## T02\_createKeyedLog—Create a Persistent Keyed Log Collection

This function creates an empty persistent keyed log collection using the default data definition (DD) and assigns a persistent identifier (PID) to the keyed log collection.

### Format

```
#include <c$to2.h>
long T02_createKeyedLog (      T02_PID_PTR   pid_ptr,
                              T02_ENV_PTR   env_ptr,
                              const long    *entryLength,
                              const long    *numberEntries,
                              const long    *keyFieldLength,
                              const long    *keyFieldDisplacement);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an element in the collection. The maximum length is 4000 bytes.

#### numberEntries

The pointer to the value that is the number of log entries that will be maintained in the log. Once the log contains this number of entries, the log will wrap and start overlaying the oldest entry with each new entry. The maximum number of entries is 2 GB of elements (where 1 GB equals 1 073 741 824).

#### keyFieldLength

The pointer to a value that contains the length of the field in the element that will be used as the key. The maximum length is 256 bytes.

#### keyFieldDisplacement

The pointer to a value that contains the displacement of the field in the element that will be used as the key.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
```

### Programming Considerations

A commit scope will be created for the T02\_createKeyedLog request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createKeyedLog request, the scope will be rolled back.

## Examples

The following example creates a persistent keyed log collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;      /* PTR to the T02 environment */
T02_PID        collect;      /* will hold collection's PID */
long           entryLength;   /* Max Entry Length */
long           numberEntries; /* Nbr Entry */
long           keyFieldLen;    /* Key field length */
long           keyFieldDisp;  /* Key field displacement */
:
:
if (T02_createKeyedLog(&collect,
                      env_ptr,
                      &entryLength,
                      &numberEntries,
                      &keyFieldLen,
                      &keyFieldDisp) == T02_ERROR)
{
    printf("T02_createKeyedLog failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createKeyedLog successful!\n");
}
```

## Related Information

- “TO2\_createKeyedLogTemp—Create a Temporary Keyed Log Collection” on page 962
- “TO2\_createKeyedLogWithOptions—Create an Empty Keyed Log Collection with Options” on page 964.

## T02\_createKeyedLogTemp—Create a Temporary Keyed Log Collection

This function creates a temporary collection for the keyed log collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createKeyedLogTemp (      T02_PID_PTR   pid_ptr,
                                   T02_ENV_PTR   env_ptr,
                                   const long    *entryLength,
                                   const long    *numberEntries,
                                   const long    *keyFieldLength,
                                   const long    *keyFieldDisplacement);
```

#### pid\_ptr

The pointer to the location where the temporary PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an element in the collection. The maximum length is 4000 bytes.

#### numberEntries

The pointer to the value that is the number of log entries to be maintained in the log. Once the log contains this number of entries, the log will wrap and start overlaying the oldest entry with each new entry.

#### keyFieldLength

The pointer to a value that contains the length of the field in the element that will be used as the key. The maximum length is 256 bytes.

#### keyFieldDisplacement

The pointer to a value that contains the displacement of the field in the element that will be used as the key.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.



- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

## Examples

The following example creates a temporary keyed log collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* PTR to the T02 environment */
T02_PID        collect;    /* Will hold collection's PID */
long           entryLength; /* Max Entry Length */
long           numberEntries; /* Nbr Entry */
long           keyFieldLen; /* Key field length */
long           keyFieldDisp; /* Key field displacement */
:
:

if (T02_createKeyedLogTemp(&collect,
                          env_ptr,
                          &entryLength,
                          &numberEntries,
                          &keyFieldLen,
                          &keyFieldDisp) == T02_ERROR)
{
    printf("T02_createKeyedLogTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createKeyedLogTemp successful!\n");
}
:
:
T02_deleteCollection(&collect, env_ptr);
```

## Related Information

- “TO2\_createKeyedLog—Create a Persistent Keyed Log Collection” on page 960
- “TO2\_createKeyedLogWithOptions—Create an Empty Keyed Log Collection with Options” on page 964
- “TO2\_deleteCollection—Delete a Collection” on page 1036.

## T02\_createKeyedLogWithOptions—Create an Empty Keyed Log Collection with Options

This function creates an empty log collection using the specified options and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createKeyedLogWithOptions (T02_PID_PTR    pid_ptr,
                                   T02_ENV_PTR     env_ptr,
                                   const T02_OPTION_PTR optionListPtr,
                                   const long      *entryLength,
                                   const long      *numberEntries,
                                   const long      *keyFieldLength,
                                   const long      *keyFieldDisplacement);
```

#### **pid\_ptr**

The pointer to the location where the PID assigned to the new collection will be returned.

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **optionListPtr**

The pointer to a returned options list from a T02\_createOptionList function call or NULL if no options are required.

#### **entryLength**

The pointer to the length of an element in the collection. The maximum length is 4000 bytes.

#### **numberEntries**

The pointer to the value that is the number of log entries that will be maintained in the log. Once the log contains this number of entries, the log will wrap and start overlaying the oldest entry with each new entry.

#### **keyFieldLength**

The pointer to a value that contains the length of the field in the element that will be used as the key. The maximum length is 256 bytes.

#### **keyFieldDisplacement**

The pointer to a value that contains the displacement of the field in the element that will be used as the key.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_DDNAME
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

## Programming Considerations

- A commit scope will be created for the T02\_createKeyedLogWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createKeyedLogWithOptions request, the scope will be rolled back.
- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

## Examples

The following example creates a persistent keyed log collection.

```
#include <c$to2.h>           /* Needed for T02 API functions      */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;     /* Pointer to the T02 environment */
T02_PID        collect;     /* will hold collection's PID      */
u_char         ddname[] = "NAME_OF_USER_DD_DEFINITION ";
long           entryLength; /* Max Entry Length                */
long           numberEntries; /* Nbr Entry                      */
long           keyFieldLen; /* Key field length                 */
long           keyFieldDisp; /* Key field displacement           */
T02_OPTION_PTR optionListPtr;
char           recoupname[] = "HOSP0001";

:

/* invoke T02 to create option list */
optionListPtr=T02_createOptionList(env_ptr,
                                   T02_OPTION_LIST_CREATE, /* create options */
                                   T02_CREATE_SHADOW,       /* use shadowing  */
                                   T02_CREATE_DD,           /* use provided dd name*/
                                   T02_CREATE_RECOUP,       /* use provided recoup */
                                   T02_OPTION_LIST_END,     /* end of options   */
                                   dd_name,                /* address of ddname */
                                   recoupname);             /* addr of recoup key */

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createKeyedLogWithOptions(&collect,
                                   env_ptr,
                                   optionListPtr,
                                   &numberEntries,
                                   &keyFieldLen,
                                   &keyFieldDisp) == T02_ERROR)
{
    printf("T02_createKeyedLogWithOptions failed!\n");
    process_error(env_ptr);
}
printf("T02_createKeyedLogWithOptions successful!\n");
```

## Related Information

- “T02\_createKeyedLog—Create a Persistent Keyed Log Collection” on page 960
- “T02\_createKeyedLogTemp—Create a Temporary Keyed Log Collection” on page 962
- “T02\_createOptionList—Create a TPF Collection Support Option List” on page 990.

## T02\_createKeySet—Create a Persistent Key Set Collection

This function creates an empty persistent key set collection using the default data definition (DD) and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createKeySet (      T02_PID_PTR    pid_ptr,
                             T02_ENV_PTR    env_ptr,
                             const long     *entryLength,
                             const long     *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 1000 bytes.

#### keyLength

The pointer to the length of the key. The maximum length is 256 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

A commit scope will be created for the T02\_createKeySet request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createKeySet request, the scope will be rolled back.

### Examples

The following example creates a persistent key set collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* PTR to the T02 environment */
T02_PID        collect;    /* will hold collection's PID */
long           entryLength; /* Max Entry Length */
long           keyLength;   /* Key field length */
:
:
if (T02_createKeySet(&collect,
                    env_ptr,
                    &entryLength,
                    &keyLength) == T02_ERROR)
```

```
{
    printf("TO2_createKeySet failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_createKeySet successful!\n");
}
```

## Related Information

- “TO2\_createKeySetTemp—Create a Temporary Key Set Collection” on page 968
- “TO2\_createKeySetWithOptions—Create an Empty Key Set Collection” on page 970.

## T02\_createKeySetTemp—Create a Temporary Key Set Collection

This function creates a temporary collection for the key set collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createKeySetTemp (      T02_PID_PTR  pid_ptr,
                                T02_ENV_PTR  env_ptr,
                                const long    *entryLength,
                                const long    *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 1000 bytes.

#### keyLength

The pointer to the length of the key. The maximum length is 256 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

### Examples

The following example creates a temporary key set collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* PTR to the T02 environment */
```

## TO2\_createKeySetTemp

```
TO2_PID      collect;      /* will hold collection's PID      */
long         entryLength;   /* Max Entry Length      */
long         keyLength;     /* Key field length      */
:
if (TO2_createKeySetTemp(&collect,
                        env_ptr,
                        &entryLength,
                        &keyLength) == TO2_ERROR)
{
    printf("TO2_createKeySetTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_createKeySetTemp successful!\n");
}
:
TO2_deleteCollection(&collect, env_ptr);
```

## Related Information

- “TO2\_createKeySet—Create a Persistent Key Set Collection” on page 966
- “TO2\_createKeySetWithOptions—Create an Empty Key Set Collection” on page 970
- “TO2\_deleteCollection—Delete a Collection” on page 1036.

## T02\_createKeySetWithOptions—Create an Empty Key Set Collection

This function creates an empty key set collection using the specified options and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createKeySetWithOptions (      T02_PID_PTR    pid_ptr,
                                         T02_ENV_PTR    env_ptr,
                                         const T02_OPTION_PTR optionListPtr,
                                         const long      *entryLength,
                                         const long      *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList function call or NULL if no options are required.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 1000 bytes.

#### keyLength

The pointer to the length of the key. The maximum length is 256 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_DDNAME
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

- A commit scope will be created for the T02\_createKeySetWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createKeySetWithOptions request, the scope will be rolled back.
- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.



## Examples

The following example creates a persistent key set collection.

```
#include <c$to2.h>                /* Needed for T02 API functions      */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment  */
T02_PID        collect;          /* will hold collection's PID      */
u_char         ddname[] = "NAME_OF_USER_DD_DEFINITION";
long           entryLength;       /* PTR to Max Entry Length        */
long           keyLength;         /* PTR to key field length         */
T02_OPTION_PTR optionListPtr;
char           recoupname[] = "HOSP0001";

:

/*      invoke T02 to create option list */
optionListPtr=T02_createOptionList(env_ptr,
                                   T02_OPTION_LIST_CREATE, /* create options      */
                                   T02_CREATE_SHADOW,       /* use shadowing       */
                                   T02_CREATE_DD,            /* use provided dd name */
                                   T02_CREATE_RECOUP,        /* use provided recoup  */
                                   T02_OPTION_LIST_END,      /* end of options      */
                                   dd_name,                  /* address of ddname    */
                                   recoupname);              /* addr of recoup key  */

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createKeyedSetWithOptions(&collect,
                                  env_ptr,
                                  optionListPtr,
                                  &entryLength,
                                  &keyLength) == T02_ERROR)
{
    printf("T02_createKeyedSetWithOptions failed!\n");
    process_error(env_ptr);
}
printf("T02_createKeyedSetWithOptions successful!\n");
```

## Related Information

- “TO2\_createKeySet—Create a Persistent Key Set Collection” on page 966
- “TO2\_createKeySetTemp—Create a Temporary Key Set Collection” on page 968
- “TO2\_createOptionList—Create a TPF Collection Support Option List” on page 990.

## T02\_createKeySortedBag—Create a Persistent Key Sorted Bag Collection

This function creates an empty persistent key sorted bag collection using the default data definition (DD) and assigns a persistent identifier (PID) to the collection.

The only difference between a key sorted bag collection and a key sorted set collection is that a key sorted bag collection has nonunique keys.

### Format

```
#include <c$to2.h>
long T02_createKeySortedBag (      T02_PID_PTR   pid_ptr,
                                  T02_ENV_PTR   env_ptr,
                                  const long     *entryLength,
                                  const long     *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID that is assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 1000 bytes.

#### keyLength

The pointer to the length of the key. For a key sorted bag collection, the maximum key length is 248 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

A commit scope will be created for the T02\_createKeySortedBag request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createKeySortedBag request, the scope will be rolled back.

### Examples

The following example creates a persistent key sorted bag collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;      /* PTR to the T02 environment */
T02_PID       collect;      /* will hold collection's PID */
long          entryLength=100; /* set to entry length */
```

## TO2\_createKeySortedBag

```
long          keyLength=8;          /* set to key length          */
:
if (TO2_createKeySortedBag(&collect,
                          env_ptr,
                          &entryLength,
                          &keylength) == T02_ERROR)
{
    printf("TO2_createKeySortedBag failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_createKeySortedBag successful!\n");
}
```

## Related Information

- “TO2\_createKeySortedBagTemp—Create a Temporary Sorted Key Bag” on page 974
- “TO2\_createKeySortedBagWithOptions—Create an Empty Key Sorted Bag with Options” on page 976.

## T02\_createKeySortedBagTemp—Create a Temporary Sorted Key Bag

This function creates a temporary key sorted bag collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createKeySortedBagTemp (      T02_PID_PTR  pid_ptr,
                                       T02_ENV_PTR  env_ptr,
                                       const long    *entryLength,
                                       const long    *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 1000 bytes.

#### keyLength

The pointer to the length of the key. The maximum key length is 248 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

### Examples

The following example creates a temporary key sorted bag collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* PTR to the T02 environment */
```

## TO2\_createKeySortedBagTemp

```
TO2_PID      collect;          /* will hold collection's PID      */
long         entryLength=100;  /* set to entry length      */
long         keyLength=8;      /* set to key length        */
:
:
if (TO2_createKeySortedBagTemp(&collect,
                               env_ptr,
                               &entryLength,
                               &keyLength) == T02_ERROR)
{
    printf("TO2_createKeySortedBagTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_createKeySortedBagTemp successful!\n");
}
:
:
TO2_deleteCollection(&collect, env_ptr);
```

## Related Information

- “TO2\_createKeySortedBag—Create a Persistent Key Sorted Bag Collection” on page 972
- “TO2\_createKeySortedBagWithOptions—Create an Empty Key Sorted Bag with Options” on page 976
- “TO2\_deleteCollection—Delete a Collection” on page 1036.

## T02\_createKeySortedBagWithOptions—Create an Empty Key Sorted Bag with Options

This function creates an empty key sorted bag collection using the specified options and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createKeySortedBagWithOptions (      T02_PID_PTR    pid_ptr,
                                              T02_ENV_PTR    env_ptr,
                                              const T02_OPTION_PTR optionListPtr,
                                              const long      *entryLength,
                                              const long      *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList call or NULL if no options are required.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 1000 bytes.

#### keyLength

The pointer to the length of the key. For a key sorted bag collection, the maximum key length is 248 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_DDNAME
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

- A commit scope will be created for the T02\_createKeySortedBagWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createKeySortedBagWithOptions request, the scope will be rolled back.
- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

## Examples

The following example creates a persistent key sorted bag collection using the specified options.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* Pointer to the T02 environment */
T02_PID        collect;    /* will hold collection's PID */
u_char         dd_name[] = "NAME_OF_USER_DD_DEFINITION" ;
long           entryLength=248; /* set to max entry length */
long           keyLength=8;    /* set to key length */
T02_OPTION_PTR optionListPtr;
char           recoupname[] = "HOSP0001";

/* invoke T02 to create option list */
:
optionListPtr=T02_createOptionList(env_ptr,
                                   T02_OPTION_LIST_CREATE, /* create options */
                                   T02_CREATE_SHADOW,      /* use shadowing */
                                   T02_CREATE_DD,          /* use provided dd name */
                                   T02_CREATE_RECOUP,       /* use provided recoup */
                                   T02_OPTION_LIST_END,     /* end of options */
                                   dd_name,                /* address of ddname */
                                   recoupname);             /* addr of recoup key */

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createKeySortedBagWithOptions(&collect,
                                       env_ptr,
                                       optionListPtr,
                                       &entryLength,
                                       &keyLength) == T02_ERROR)
{
    printf("T02_createKeySortedBagWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createKeySortedBagWithOptions successful!\n");
}
```

## Related Information

- “T02\_createKeySortedBag—Create a Persistent Key Sorted Bag Collection” on page 972
- “T02\_createKeySortedBagTemp—Create a Temporary Sorted Key Bag” on page 974
- “T02\_createOptionList—Create a TPF Collection Support Option List” on page 990.

## T02\_createKeySortedSet—Create an Empty Persistent Key Sorted Set Collection

This function creates an empty persistent key sorted set collection (also known as a dictionary) using the default data definition (DD) and assigns a persistent identifier (PID) to the key sorted set collection.

### Format

```
#include <c$to2.h>
long T02_createKeySortedSet (      T02_PID_PTR   pid_ptr,
                                  T02_ENV_PTR   env_ptr,
                                  const long     *entryLength,
                                  const long     *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 1000 bytes. For a key sorted set, all entries must be equal to or less than the maximum length.

#### keyLength

The pointer to the length of the key. The maximum length is 256 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

A commit scope will be created for the T02\_createKeySortedSet request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createKeySortedSet request, the scope will be rolled back.

### Examples

The following example creates a persistent key sorted set collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;           /* PTR to the T02 environment */
T02_PID_PTR   collect;           /* will hold collection's PID */
long          entryLength=100;   /* set to entry length */
long          keylength=8;       /* set to key length */
```



```

:
if (TO2_createKeySortedSet(&collect, env_ptr,
                           &entryLength,
                           &keylength) == T02_ERROR)
{
    printf("TO2_createKeySortedSet failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_createKeySortedSet successful!\n");
}

```

## Related Information

- “TO2\_createKeySortedSetTemp—Create a Temporary Key Sorted Set” on page 980
- “TO2\_createKeySortedSetWithOptions—Create an Empty Key Sorted Set Collection” on page 982.

## T02\_createKeySortedSetTemp—Create a Temporary Key Sorted Set

This function creates a temporary key sorted set collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createKeySortedSetTemp (      T02_PID_PTR  pid_ptr,
                                       T02_ENV_PTR  env_ptr,
                                       const long    *entryLength,
                                       const long    *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 1000 bytes.

#### keyLength

The pointer to the length of the key. The maximum length is 256 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

### Examples

The following example creates a temporary key sorted set collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* PTR to the T02 environment */
```

## T02\_createKeySortedSetTemp

```
T02_PID      collect;          /* will hold collection's PID      */
long         entryLength=100;  /* set to entry length      */
long         keylength=8;      /* set to key length        */
:
:
if (T02_createKeySortedSetTemp(&collect,
                               env_ptr,
                               &entryLength,
                               &keylength) == T02_ERROR)
{
    printf("T02_createKeySortedSetTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createKeySortedSetTemp successful!\n");
}
:
T02_deleteCollection(&collect, env_ptr);
```

## Related Information

- “T02\_createKeySortedSet—Create an Empty Persistent Key Sorted Set Collection” on page 978
- “T02\_createKeySortedSetWithOptions—Create an Empty Key Sorted Set Collection” on page 982
- “T02\_deleteCollection—Delete a Collection” on page 1036.

## T02\_createKeySortedSetWithOptions—Create an Empty Key Sorted Set Collection

This function creates an empty key sorted set collection using the specified options and assigns a persistent identifier (PID) to the key sorted set collection.

### Format

```
#include <c$to2.h>
long T02_createKeySortedSetWithOptions (      T02_PID_PTR    pid_ptr,
                                              T02_ENV_PTR    env_ptr,
                                              const T02_OPTION_PTR optionListPtr,
                                              const long      *entryLength,
                                              const long      *keyLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList call or NULL if no options are required.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 1000 bytes.

#### keyLength

The pointer to the length of the key. The maximum length is 248 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_DDNAME
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

- A commit scope will be created for the T02\_createKeySortedSetWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createKeySortedSetWithOptions request, the scope will be rolled back.
- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

## Examples

The following example creates a persistent key sorted set collection using the specified options.

```
#include <c$to2.h>          /* Needed for T02 API functions      */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* Pointer to the T02 environment */
T02_PID        collect;    /* will hold collection's PID      */
u_char         dd_name[] = "NAME_OF_USER_DD_DEFINITION";
long           entryLength=248; /* set to max entry length      */
long           keylength=8;   /* set to key length              */
T02_OPTION_PTR optionListPtr;
char           recoupname[] = "HOSP0001";

                                /* invoke T02 to create option list */
                                :
                                :
optionListPtr=T02_createOptionList(env_ptr,
                                   T02_OPTION_LIST_CREATE, /* create options */
                                   T02_CREATE_SHADOW,      /* use shadowing  */
                                   T02_CREATE_DD,          /* use provided dd name*/
                                   T02_CREATE_RECOUP,      /* use provided recoup */
                                   T02_OPTION_LIST_END,    /* end of options   */
                                   dd_name,                /* address of ddname */
                                   recoupname);            /* addr of recoup key */

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createKeySortedSetWithOptions(&collect,
                                       env_ptr,
                                       optionListPtr,
                                       &entryLength,
                                       &keylength) == T02_ERROR)
{
    printf("T02_createKeySortedSetWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createKeySortedSetWithOptions successful!\n");
}
```

## Related Information

- “T02\_createKeySortedSet—Create an Empty Persistent Key Sorted Set Collection” on page 978
- “T02\_createKeySortedSetTemp—Create a Temporary Key Sorted Set” on page 980
- “T02\_createOptionList—Create a TPF Collection Support Option List” on page 990.

## T02\_createLog—Create an Empty Persistent Log Collection

This function creates an empty persistent log collection using the default data definition (DD) and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createLog (      T02_PID_PTR    pid_ptr,
                        T02_ENV_PTR    env_ptr,
                        const long      *entryLength,
                        const long      *numberEntries);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an element in the collection. The maximum length is 4000 bytes.

#### numberEntries

The pointer to the value that is the number of log entries to be maintained in the log. Once the log contains this number of entries, the log will wrap and start overlaying the oldest entry with each new entry.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
```

### Programming Considerations

A commit scope will be created for the T02\_createLog request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createLog request, the scope will be rolled back.

### Examples

The following example creates a persistent log collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* PTR to the T02 environment */
T02_PID        collect;    /* will hold collection's PID */
long           entryLength; /* Max Entry Length */
long           numberEntries; /* Number Entry */
:
:
if (T02_createLog(&collect,
                 env_ptr,
                 &entryLength,
```

```
        &numberEntries) == T02_ERROR)
{
    printf("TO2_createLog failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_createLog successful!\n");
}
```

## Related Information

- “TO2\_createLogTemp—Create a Temporary Log Collection” on page 986
- “TO2\_createLogWithOptions—Create an Empty Log Collection with Options” on page 988.

## T02\_createLogTemp—Create a Temporary Log Collection

This function creates a temporary collection for the log collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createLogTemp (      T02_PID_PTR   pid_ptr,
                             T02_ENV_PTR   env_ptr,
                             const long    *entryLength,
                             const long    *numberEntries);
```

#### pid\_ptr

The pointer to the location where the temporary PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an element in the collection. The maximum length is 4000 bytes.

#### numberEntries

The pointer to the value that is the number of log entries to be maintained in the log. Once the log contains this number of entries, the log will wrap and start overlaying the oldest entry with each new entry.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

### Examples

The following example creates a temporary log collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */
```



## TO2\_createLogTemp

```
TO2_ENV_PTR    env_ptr;        /* PTR to the T02 environment    */
TO2_PID        collect;        /* will hold collection's PID    */
long           entryLength;     /* Max Entry Length              */
long           numberEntries;   /* Number Entry                   */
:
:
if (TO2_createLogTemp(&collect,
                     env_ptr,
                     &entryLength,
                     &numberEntries) == T02_ERROR)
{
    printf("TO2_createLogTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_createLogTemp successful!\n");
}
:
:
TO2_deleteCollection(&collect, env_ptr);
```

## Related Information

- “TO2\_createLogTemp—Create a Temporary Log Collection” on page 986
- “TO2\_createLogWithOptions—Create an Empty Log Collection with Options” on page 988
- “TO2\_deleteCollection—Delete a Collection” on page 1036.

## T02\_createLogWithOptions—Create an Empty Log Collection with Options

This function creates an empty log collection using the specified options and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createLogWithOptions (      T02_PID_PTR    pid_ptr,
                                     T02_ENV_PTR    env_ptr,
                                     const T02_OPTION_PTR optionListPtr,
                                     const long      *entryLength,
                                     const long      *numberEntries);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList function call or NULL if no options are required.

#### entryLength

The pointer to the length of an element in the collection. The maximum length is 4000 bytes.

#### numberEntries

The pointer to the value that is the number of log entries to be maintained in the log. Once the log contains this number of entries, the log will wrap and start overlaying the oldest entry with each new entry.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_DDNAME
T02_ERROR_ENV
```

### Programming Considerations

- A commit scope will be created for the T02\_createLogWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createLogWithOptions request, the scope will be rolled back.
- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

## Examples

The following example creates a persistent log collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
T02_PID        collect;         /* will hold collection's PID */
u_char         ddname[] = "NAME_OF_USER_DD_DEFINITION ";
long           entryLength;      /* Max Entry Length */
T02_OPTION_PTR optionListPtr;
char           recoupname[] = "HOSP0001";
long           numberEntries;    /* Number Entry */

:

/* invoke T02 to create option list */
optionListPtr=T02_createOptionList(env_ptr,
                                   T02_OPTION_LIST_CREATE, /* create options */
                                   T02_CREATE_SHADOW,      /* use shadowing */
                                   T02_CREATE_DD,          /* use provided dd name*/
                                   T02_CREATE_RECOUP,      /* use provided recoup */
                                   T02_OPTION_LIST_END,    /* end of options */
                                   dd_name,               /* address of ddname */
                                   recoupname);           /* addr of recoup key */

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createLogWithOptions(&collect,
                             env_ptr,
                             optionListPtr,
                             &entryLength,
                             &numberEntries) == T02_ERROR)
{
    printf("T02_createLogWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createLogWithOptions successful!\n");
}
```

## Related Information

- “TO2\_createLog—Create an Empty Persistent Log Collection” on page 984
- “TO2\_createLogTemp—Create a Temporary Log Collection” on page 986
- “TO2\_createOptionList—Create a TPF Collection Support Option List” on page 990.

## T02\_createOptionList—Create a TPF Collection Support Option List

This function creates an option list structure that can be passed to succeeding function calls that take an option list as an input parameter.

### Format

```
#include <c$to2.h>
T02_OPTION_PTR T02_createOptionList (T02_ENV_PTR      env_ptr,
                                     enum T02_OPTION_LIST_TYPE optionType,
                                     enum T02_OPTION_NAME  option1,
                                     enum T02_OPTION_NAME  option2,
                                     ...optionx,
                                     enum T02_OPTION_LIST_TYPE endFlag,
                                     void                  *option1value,
                                     void                  *option2value,
                                     ...*optionyvalue);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### optionType

Defines the type of option list to build.

#### T02\_OPTION\_LIST\_CREATE

Build a collection option list for use with the T02\_create...WithOptions, T02\_restore, and T02\_copyCollection functions.

#### T02\_OPTION\_LIST\_DD

Build a data definition (DD) option list for use with the T02\_changedDD and T02\_createDD functions.

#### T02\_OPTION\_LIST\_DS

Build a data store (DS) option list for use with the T02\_changedDS and T02\_createDS functions.

#### option1

#### option2

⋮

#### optionx...

The first, second, and repeat options. Repeat the options until all the options that you want are listed.

**Note:** The T02\_CREATE... options are used with the create option list (T02\_OPTION\_LIST\_CREATE).

#### T02\_CREATE\_DD

Use the DD name provided as an option value. The DD name must be 32 characters.

#### T02\_CREATE\_FORCE

Force the collection to be created to have extended residency.

#### T02\_CREATE\_NOFORCE

Do not force the collection to be created to have extended residency.

#### T02\_CREATE\_NOSHADOW

Do not use shadowing when the collection is created.

#### T02\_CREATE\_NULL

Placeholder for an application program that uses its own parameter list to determine which options to specify.

**TO2\_CREATE\_RECOUP**

Associate the recoup index name provided as an option value with the collection to be created. The recoup index name must be 8 characters.

**TO2\_CREATE\_SHADOW**

Use shadowing when the collection is created.

**TO2\_CREATE\_TEMPORARY**

Create a temporary collection.

**Note:** The T02\_DD... options are used with the DD option list (TO2\_OPTION\_LIST\_DD).

**TO2\_DD\_DATARID**

Define the record ID for an internal data record. The expected option value is a pointer to a 2-character field, which contains the hexadecimal record ID to be assigned to the data records.

**TO2\_DD\_DIRECTRID**

Define the record ID for an internal directory record. The expected option value is a pointer to a 2-character field, which contains the hexadecimal record ID to be assigned to the directory records.

**TO2\_DD\_FORCE**

Build the collection using an extended structure rather than the normal compact structure.

**TO2\_DD\_INDEXRID**

Define the record ID for an internal index record. The expected option value is a pointer to a 2-character field, which contains the hexadecimal record ID to be assigned to the index records.

**TO2\_DD\_NOFORCE**

Build the collection initially with normal compact structure.

**TO2\_DD\_NOSHADOW**

Do not use shadowing.

**TO2\_DD\_NULL**

Placeholder for an application program that uses its own parameter list to determine which options to specify.

**TO2\_DD\_SHADOW**

Use shadowing.

**TO2\_DD\_TEMP**

The collection created will use the ddname provided.

**Note:** The T02\_DS... options are used with the DS option list (TO2\_OPTION\_LIST\_DS).

**TO2\_DS\_DELETE\_DELAY**

Mark the collection for deletion, but delay the actual deletion to allow time for the collection to be reclaimed at a later time. The system delay time is set at 48 hours. After this time, the collection will actually be deleted and its resources returned to the system.

**TO2\_DS\_DELETE\_IMMED**

Immediately delete collections at the time of the delete request.

**TO2\_DS\_INVENTORY**

Build the data store with a persistent identifier (PID) inventory collection. If a data store contains a PID inventory, TPF collection support will store the

## T02\_createOptionList

PID of every collection created for the data store in the inventory collection of the data store. When the collection is deleted, the PID of the collection will be deleted from the inventory of the data store.

### TO2\_DS\_NOINVENTORY

Build the data store with no PID inventory collection or delete the inventory collection if it already exists.

### TO2\_DS\_NULL

Placeholder for an application program that uses its own parameter list to determine which options to specify.

### endFlag

Marks the end of the list of options.

### TO2\_OPTION\_LIST\_END

End of option list.

### option1value

### option2value

⋮

### optionyvalue

The information for the first, second, and repeat options that take variable values. TO2\_CREATE\_DD, TO2\_CREATE\_RECOUP, TO2\_DD\_DATARID, TO2\_DD\_DIRECTRID, and TO2\_DD\_INDEXRID all need corresponding option values. Repeat the option values in the order that the options were specified until all required option values have been specified.

## Normal Return

A pointer to an option list structure to be passed on the associated T02\_...WithOptions request.

## Error Return

An error return is indicated by a NULL return code. When NULL is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

TO2\_ERROR\_OPTION

TO2\_ERROR\_OPTION\_CONFLICTS

## Programming Considerations

Once you have completed using the returned option list structure, you must release the area pointed to by the option list pointer by using the free function.

## Examples

The following is an example of how to use a T02\_createOptionList function to create an option list to use as a parameter on a T02\_createBLOBWithOptions request.

```
#include <c$to2.h>                                /* Needed for T02 API functions */

T02_ENV_PTR   env_ptr;                            /* Pointer to the T02 environment */
T02_PID       collect;                            /* will hold collection's PID */
T02_OPTION_PTR optionListPtr;
long          entryLength=248;                    /* set to max entry length */
long          keylength=8;                        /* set to key length */
```

## T02\_createOptionList

```

/* name of the DD to use for definition */
char      ddname[] = "HOSPITAL_PATIENT_LIST";
char      recoupname[] = "HOSP0001";

/* invoke T02 to create option list */
:
optionListPtr=T02_createOptionList(env_ptr,
                                  T02_OPTION_LIST_CREATE, /* create options */
                                  T02_CREATE_SHADOW,       /* use shadowing */
                                  T02_CREATE_DD,           /* use provided dd name*/
                                  T02_CREATE_RECOUP,       /* use provided recoup */
                                  T02_OPTION_LIST_END,     /* end of options */
                                  ddname,                 /* address of ddname */
                                  recoupname);             /* addr of recoup key */

if (optionListPtr == T02_ERROR)
    reportError(env_ptr);

/* invoke T02 create dictionary using option list */
if (T02_createDictionaryWithOptions(&collect,
                                    env_ptr,
                                    optionListPtr
                                    &entryLength
                                    &keyLength) == T02_ERROR)
    reportError(env_ptr);
:
free(optionListPtr); /* release option list area */

```

The following example shows how to use the T02\_OPTION\_NULL value to code a dynamic T02\_createOptionList function call. Because not all calls will require all options, T02\_OPTION\_NULL allows the application to use a single T02\_createOptionList call to generate multiple different option lists depending on the parameters of the caller.

```

/*****
/* start generateOptionList function */
*****/
#include <c$to2.h> /* Needed for T02 API functions */
#include <stdio.h> /* APIs for standard I/O functions */

T02_OPTION_PTR generateOptionList( /*
                                T02_ENV_PTR env_ptr,
                                char shadow[2],
                                char ddname[NAME_LGH],
                                char recoupIndex[RECOUPINDEX_LGH])
{
    T02_OPTION_PTR optionList_ptr=NULL;
    /*
    /* generate an array of options and set each option to
    /* T02_CREATE_NULL in case it is not needed for this call.
    /*
    long option[4]={T02_CREATE_NULL, T02_CREATE_NULL,
                   T02_CREATE_NULL, T02_CREATE_NULL};
    char * pointer[2]={0, 0};
    long i=0, j=0;

    /*
    /* The following code works through the caller's parameters
    /* and sets up the correct array elements for the options
    /* specified.
    /*
    :
    :
    if ((shadow[0]==1) && (shadow[1]=='Y')) {
        option[i]=T02_CREATE_SHADOW;
        i++;
    }
}

```

## T02\_createOptionList

```
if ((long)ddname != 0) {
    option[i]=T02_CREATE_DD;
    pointer[j]=ddname;
    i++; j++;
}
if ((long)recoupIndex != 0) {
    option[i]=T02_CREATE_RECOUP;
    pointer[j]=recoupIndex;
    i++; j++;
}
optionList_ptr = T02_createOptionList(env_ptr,
                                     T02_OPTION_LIST_CREATE,
                                     option[0],
                                     option[1],
                                     option[2],
                                     option[3],
                                     T02_OPTION_LIST_END,
                                     pointer[0],
                                     pointer[1]);

if ((long)optionList_ptr == NULL) {
    printf("T02_createOptionList failed!\n");
    process_error(env_ptr);
    return NULL;
}
return optionList_ptr;
}
```

## Related Information

None.



---

## **TO2\_createOrder—Create a Persistent Ordered Collection**

This function has been renamed to T02\_createSequence. For more information, see “TO2\_createSequence—Create a Persistent Sequence Collection” on page 1000.

## **T02\_createOrderTemp—Create a Temporary Ordered Collection**

This function has been renamed to `T02_createSequenceTemp`. For more information, see “`T02_createSequenceTemp—Create a Temporary Sequence Collection`” on page 1002.

---

## **T02\_createOrderWithOptions—Create an Empty Ordered Collection with Options**

This function has been renamed to T02\_createSequenceWithOptions. For more information, see “T02\_createSequenceWithOptions—Create an Empty Sequence Collection with Options” on page 1004.

## T02\_createRecoupIndex—Create a Recoup Index

This function creates a TPF collection support (TPFCS) recoup index.

### Format

```
#include <c$to2.h>
long T02_createRecoupIndex (      T02_ENV_PTR          env_ptr,
                                const void             *indexName,
                                const enum T02_RECOUP_TYPE collectionType,
                                const enum T02_RECOUP_CONTROL control,
                                const char             userProgramName[],
                                const char             functionName[],
                                const char             description[]);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### indexName

The pointer to the 8-byte recoup index name for the recoup index that will be created.

#### collectionType

A pointer to the collection type.

##### TO2\_RECOUP\_IGNORED

The collection is a binary large object (BLOB), which indicates that the recoup index is not homogeneous or heterogeneous.

##### TO2\_RECOUP\_HOMOGENEOUS

All elements in the collection have the same displacement to the persistent identifier (PID) or file address and can be recouped the same way.

##### TO2\_RECOUP\_HETEROGENEOUS

Not all elements contain an embedded file address or PID, or the displacements are different (that is, not all elements can be recouped the same way).

#### control

An indicator to which processing restrictions apply.

##### TO2\_RECOUP\_CONTROL\_NONE

No specific control instructions.

#### userProgramName

A pointer to the user exit program name, if TO2\_RECOUP\_CONTROL\_USER was specified, otherwise NULL is passed. **userProgramName** can be up to 4 bytes.

#### functionName

The name of the function in the user program to activate if **TO2\_RECOUP\_CONTROL\_USER** was specified, otherwise NULL and the program never will be activated.

#### description

The description of the recoup index can be up to 256 bytes.

### Normal Return

The normal return is a positive value.

## Error Return

An error return is indicated by a zero. When zero is returned, use the `TO2_getErrorCode` function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

`TO2_ERROR_ENV`

## Programming Considerations

The index cannot be shared by different structures; for example, between bag and key sorted bag collections.

## Examples

The following example creates a homogeneous recoup index.

```
#include <c$to2.h>           /* Needed for TO2 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

TO2_PID      collect;        /* target object's PID */
TO2_ENV_PTR  env_ptr;        /* Pointer to TO2 environment */
TO2_RECOUP_TYPE  collectionType = TO2_RECOUP_HOMOGENEOUS;
TO2_RECOUP_CONTROL  control = TO2_RECOUP_CONTROL_NONE;
/* Processing restrictions */
u_char      indexName[]="INDEX003"; /* index identifier */

:

if (TO2_createRecoupIndex(env_ptr,
                          indexName,
                          collectionType,
                          control,
                          userProgramName,
                          functionName,
                          description) == TO2_ERROR)
{
    printf ("TO2_createRecoupIndex failed!\n");
    process_error(env_ptr);
}
else
    printf("TO2_createRecoupIndex successful!\n");
```

## Related Information

- “TO2\_addRecoupIndexEntry–Add an Entry to a Recoup Index” on page 893
- “TO2\_associateRecoupIndexWithPID–Create a PID to Index Association” on page 899
- “TO2\_deleteRecoupIndex–Delete a Recoup Index” on page 1040.

## T02\_createSequence—Create a Persistent Sequence Collection

This function creates an empty persistent sequence collection using the default data definition (DD) and assigns a persistent identifier (PID) to the sequence collection.

### Format

```
#include <c$to2.h>
long T02_createSequence (      T02_PID_PTR  pid_ptr,
                               T02_ENV_PTR  env_ptr,
                               const long    *entryLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an element in the collection. The maximum length is 4000 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
```

## Programming Considerations

A commit scope will be created for the T02\_createSequence request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createSequence request, the scope will be rolled back.

## Examples

The following example creates a persistent sequence collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* PTR to the T02 environment */
T02_PID      collect;      /* will hold collection's PID */
long         entryLength;  /* Max Entry Length */

if (T02_createSequence(&collect,
                      env_ptr,
                      &entryLength) == T02_ERROR)
{
    printf("T02_createSequence failed!\n");
    process_error(env_ptr);
}
```

```
else
{
    printf("TO2_createSequence successful!\n");
}
```

## Related Information

- “TO2\_createSequenceTemp—Create a Temporary Sequence Collection” on page 1002
- “TO2\_createSequenceWithOptions—Create an Empty Sequence Collection with Options” on page 1004.

## T02\_createSequenceTemp—Create a Temporary Sequence Collection

This function creates a temporary sequence collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createSequenceTemp (      T02_PID_PTR   pid_ptr,
                                   T02_ENV_PTR   env_ptr,
                                   const long     *entryLength);
```

#### pid\_ptr

The pointer to the location where the temporary PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an element in the collection. The maximum length is 4000 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

### Examples

The following example creates a temporary sequence collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;     /* PTR to the T02 environment */
T02_PID       collect;     /* will hold collection's PID */
long          entryLength; /* PTR to Max Entry Length */
:
:
if (T02_createSequenceTemp(&collect,
                          env_ptr,
```



## TO2\_createSequenceTemp

```
                                &entryLength) == T02_ERROR)
{
    printf("TO2_createSequenceTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_createSequenceTemp successful!\n");
}
:
T02_deleteCollection(&collect, env_ptr);
```

## Related Information

- “TO2\_createSequence—Create a Persistent Sequence Collection” on page 1000
- “TO2\_createSequenceWithOptions—Create an Empty Sequence Collection with Options” on page 1004
- “TO2\_deleteCollection—Delete a Collection” on page 1036.

## T02\_createSequenceWithOptions—Create an Empty Sequence Collection with Options

This function creates an empty sequence collection using the specified options and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createSequenceWithOptions (      T02_PID_PTR    pid_ptr,
                                          T02_ENV_PTR    env_ptr,
                                          const T02_OPTION_PTR optionListPtr,
                                          const long      *entryLength);
```

#### **pid\_ptr**

The pointer to the location where the PID assigned to the new collection will be returned.

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **optionListPtr**

The pointer to a returned options list from a T02\_createOptionList function call or NULL if no options are required.

#### **entryLength**

The pointer to the length of an element in the collection. The maximum length is 4000 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_DDNAME
T02_ERROR_ENV
```

### Programming Considerations

- A commit scope will be created for the T02\_createSequenceWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createSequenceWithOptions request, the scope will be rolled back.
- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

### Examples

The following example creates a persistent sequence collection with options.

## TO2\_createSequenceWithOptions

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <c$to2.h>                /* Needed for T02 API functions */

T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
T02_PID        collect;          /* will hold collection's PID */
u_char         dd_name[] = "NAME_OF_USER_DD_DEFINITION ";
long           entryLength;      /* PTR to Max Entry Length */
T02_OPTION_PTR optionListPtr;
char           recoupname[] = "HOSP0001";

:

/* invoke T02 to create option list */
optionListPtr=T02_createOptionList(env_ptr,
    T02_OPTION_LIST_CREATE, /* create options */
    T02_CREATE_SHADOW,     /* use shadowing */
    T02_CREATE_DD,         /* use provided ddnm */
    T02_CREATE_RECOUP,     /* use provided recp */
    T02_OPTION_LIST_END,   /* end of options */
    dd_name,               /* address of ddname */
    recoupname);           /* addr of recoup key*/

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createSequenceWithOptions(&collect,
    env_ptr,
    optionListPtr,
    &entryLength) == T02_ERROR)
{
    printf("T02_createSequenceWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createSequenceWithOptions successful!\n");
}
```

## Related Information

- “TO2\_createOptionList—Create a TPF Collection Support Option List” on page 990
- “TO2\_createSequence—Create a Persistent Sequence Collection” on page 1000
- “TO2\_createSequenceTemp—Create a Temporary Sequence Collection” on page 1002.

## T02\_createSet—Create a Persistent Set Collection

This function creates an empty persistent set collection using the default data definition (DD) and assigns a persistent identifier (PID) to the set collection.

### Format

```
#include <c$to2.h>
long T02_createSet (      T02_PID_PTR   pid_ptr,
                        T02_ENV_PTR   env_ptr,
                        const long    *entryLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 256 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
```

## Programming Considerations

A commit scope will be created for the T02\_createSet request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createSet request, the scope will be rolled back.

## Examples

The following example creates a persistent set collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;     /* PTR to the T02 environment */
T02_PID       collect;     /* will hold collection's PID */
long          entryLength; /* Max Entry Length */
:
:
if (T02_createSet(&collect,
                 env_ptr,
                 &entryLength) == T02_ERROR)
{
    printf("T02_createSet failed!\n");
    process_error(env_ptr);
}
```

```
else
{
    printf("TO2_createSet successful!\n");
}
```

## Related Information

- “TO2\_createSetTemp—Create a Temporary Set for the Collection” on page 1008
- “TO2\_createSetWithOptions—Create an Empty Set Collection with Options” on page 1010.

## T02\_createSetTemp—Create a Temporary Set for the Collection

This function creates a temporary collection for the set collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createSetTemp (      T02_PID_PTR   pid_ptr,
                             T02_ENV_PTR   env_ptr,
                             const long    *entryLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 256 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

### Examples

The following example creates a temporary set collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;     /* PTR to the T02 environment */
T02_PID       collect;     /* will hold collection's PID */
long          entryLength; /* Max Entry Length */
:
:
if (T02_createSetTemp(&collect,
                    env_ptr,
```

```
        &entryLength) == T02_ERROR)
{
    printf("T02_createSetTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createSetTemp successful!\n");
}
:
T02_deleteCollection(&collect, env_ptr);
```

## Related Information

- “T02\_createSet—Create a Persistent Set Collection” on page 1006
- “T02\_createSetWithOptions—Create an Empty Set Collection with Options” on page 1010
- “T02\_deleteCollection—Delete a Collection” on page 1036.

## T02\_createSetWithOptions–Create an Empty Set Collection with Options

This function creates an empty set collection using the specified options and assigns a persistent identifier (PID) to the set collection.

### Format

```
#include <c$to2.h>
long T02_createSetWithOptions (      T02_PID_PTR    pid_ptr,
                                     T02_ENV_PTR    env_ptr,
                                     const T02_OPTION_PTR optionListPtr,
                                     const long      *entryLength);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList function call or NULL if no options are required.

#### entryLength

The pointer to the length of an entry in the collection. The maximum length is 256 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_DDNAME
T02_ERROR_ENV
```

### Programming Considerations

- A commit scope will be created for the T02\_createSetWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createSetWithOptions request, the scope will be rolled back.
- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

### Examples

The following example creates a persistent set collection.

```
#include <c$to2.h>                                /* Needed for T02 API functions */
#include <stdio.h>                                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;                          /* Pointer to the T02 environment */
```



## TO2\_createSetWithOptions

```
TO2_PID      collect;          /* will hold collection's PID      */
u_char       ddname[] = "NAME_OF_USER_DD_DEFINITION ";
long         entryLength;      /* Max Entry Length      */
TO2_OPTION_PTR optionListPtr;
char         recoupname[] = "HOSP0001";

:

/* invoke TO2 to create option list */
optionListPtr TO2_createOptionList(env_ptr,
    TO2_OPTION_LIST_CREATE, /* create options */
    TO2_CREATE_SHADOW,     /* use shadowing */
    TO2_CREATE_DD,         /* use provided ddnm */
    TO2_CREATE_RECOUP,     /* use provided recp */
    TO2_OPTION_LIST_END,   /* end of options */
    dd_name,               /* address of ddname */
    recoupname);           /* addr of recoup key*/

if (optionListPtr == TO2_ERROR)
    process_error(env_ptr);

if (TO2_createSetWithOptions(&collect,
    env_ptr,
    optionListPtr,
    &entryLength) == TO2_ERROR)
{
    printf("TO2_createSetWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_createSetWithOptions successful!\n");
}
```

## Related Information

- “TO2\_createOptionList—Create a TPF Collection Support Option List” on page 990
- “TO2\_createSet—Create a Persistent Set Collection” on page 1006
- “TO2\_createSetTemp—Create a Temporary Set for the Collection” on page 1008.

## **T02\_createSort—Create a Persistent Sorted Collection**

This function has been renamed to T02\_createSortedBag. For more information, see “T02\_createSortedBag—Create an Empty Persistent Sorted Bag Collection” on page 1013.

## TO2\_createSortedBag—Create an Empty Persistent Sorted Bag Collection

This function creates an empty persistent sorted bag collection using the default data definition (DD) and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long TO2_createSortedBag (      T02_PID_PTR  pid_ptr,
                                T02_ENV_PTR  env_ptr,
                                const long    *entryLength,
                                const long    *sortFieldLength,
                                const long    *sortFieldDisplacement);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the maximum length of an entry in the collection. The maximum length is 1000 bytes.

#### sortFieldLength

The pointer to the length of the sort field. The maximum length is 248 bytes. The **sortFieldLength** value must be equal to or less than the **entryLength** value.

#### sortFieldDisplacement

The pointer to the displacement in the entry to the start of the sort field. The **sortFieldLength** value plus the **sortFieldDisplacement** value must be equal to or less than the **entryLength** value.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
```

### Programming Considerations

A commit scope will be created for the T02\_createSortedBag request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createSortedBag request, the scope will be rolled back.

### Examples

The following example creates a persistent sorted bag collection.

## T02\_createSortedBag

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;       /* PTR to the T02 environment */
T02_PID collect;          /* will hold collection's PID */
long entryLength;         /* Max Entry Length */
long sortFieldLength;     /* Field length */
long sortFieldLengthDisplacement; /* PTR to keyFLD */
:
:
if (T02_createSortedBag(&collect,
                        env_ptr,
                        &entryLength,
                        &sortFieldLength,
                        &sortFieldLengthDisplacement) == T02_ERROR)
{
    printf("T02_createSortedBag failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createSortedBag successful!\n");
}
```

## Related Information

- “T02\_createSortedBagTemp—Create a Temporary Sorted Bag Collection” on page 1015
- “T02\_createSortedBagWithOptions—Create an Empty Sorted Bag Collection with Options” on page 1017.

## T02\_createSortedBagTemp—Create a Temporary Sorted Bag Collection

This function creates a temporary collection for the sorted bag collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

### Format

```
#include <c$to2.h>
long T02_createSortedBagTemp (      T02_PID_PTR   pid_ptr,
                                   T02_ENV_PTR   env_ptr,
                                   const long     *entryLength,
                                   const long     *sortFieldLength,
                                   const long     *sortFieldDisplacement);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the maximum length of an entry in the collection. The maximum length is 1000 bytes.

#### sortFieldLength

The pointer to the length of the sort field. The maximum length is 248 bytes. The **sortFieldLength** value must be equal to or less than the **entryLength** value.

#### sortFieldDisplacement

The pointer to the displacement in the entry to the start of the sort field. The **sortFieldLength** value plus the **sortFieldDisplacement** value must be equal to or less than the **entryLength** value.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

## T02\_createSortedBagTemp

### Examples

The following example creates a temporary sorted bag collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;       /* PTR to the T02 environment */
T02_PID collect;          /* will hold collection's PID */
long entryLength;         /* Max Entry Length */
long sortFieldLength;     /* Sort field length */
long sortFieldLengthDisplacement; /* Sort field displacement */
:
:
if (T02_createSortedBagTemp(&collect,
                           env_ptr,
                           &entryLength,
                           &sortFieldLength,
                           &sortFieldDisplacement) == T02_ERROR)
{
    printf("T02_createSortedBagTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createSortedBagTemp successful!\n");
}
:
:
T02_deleteCollection(&collect, env_ptr);
```

### Related Information

- “T02\_createSortedBag—Create an Empty Persistent Sorted Bag Collection” on page 1013
- “T02\_createSortedBagWithOptions—Create an Empty Sorted Bag Collection with Options” on page 1017
- “T02\_deleteCollection—Delete a Collection” on page 1036.

## T02\_createSortedBagWithOptions—Create an Empty Sorted Bag Collection with Options

This function creates an empty sorted bag collection using the specified options and assigns a persistent identifier (PID) to the collection.

### Format

```
#include <c$to2.h>
long T02_createSortedBagWithOptions ( T02_PID_PTR    pid_ptr,
                                     T02_ENV_PTR    env_ptr,
                                     const T02_OPTION_PTR optionListPtr,
                                     const long      *entryLength,
                                     const long      *sortFieldLength,
                                     const long      *sortFieldDisplacement);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList function call or NULL if no options are required.

#### entryLength

The pointer to the maximum length of an entry in the collection. The maximum length is 1000 bytes.

#### sortFieldLength

The pointer to the length of the sort field. The maximum length is 248 bytes. The **sortFieldLength** value must be equal to or less than the **entryLength** value.

#### sortFieldDisplacement

The pointer to the displacement in the entry to the start of the sort field. The **sortFieldLength** value plus the **sortFieldDisplacement** value must be equal to or less than the **entryLength** value.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
```

### Programming Considerations

- A commit scope will be created for the T02\_createSortedBagWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createSortedBagWithOptions request, the scope will be rolled back.

## T02\_createSortedBagWithOptions

- This request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

## Examples

The following example creates a persistent sorted bag collection.

```
#include <c$to2.h> /* Needed for T02 API funcs */
#include <stdio.h> /* APIs for standard I/O funcs */

T02_ENV_PTR env_ptr; /* Pointer to the T02 environ. */
T02_PID collect; /* will hold collection's PID */
u_char ddname[] = "NAME_OF_USER_DD_DEFINITION ";
long entryLength; /* Max Entry Length */
long sortFieldLength; /* Field length */
long sortFieldDisplacement; /* Sort key field disp. */
T02_OPTION_PTR optionListPtr;
char recoupname[] = "HOSP0001";

:
/* invoke T02 to create opt list */
optionListPtr=T02_createOptionList(env_ptr,
    T02_OPTION_LIST_CREATE, /* create options */
    T02_CREATE_SHADOW, /* use shadowing */
    T02_CREATE_DD, /* use provided ddname*/
    T02_CREATE_RECOUP, /* use provided recoup*/
    T02_OPTION_LIST_END, /* end of options */
    dd_name, /* address of ddname */
    recoupname); /* addr of recoup key */

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createSortedBagWithOptions(&collect,
    env_ptr,
    optionListPtr,
    &entryLength,
    &sortFieldLength,
    &sortFieldDisplacement) == T02_ERROR)
{
    printf("T02_createSortedBagWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createSortedBagWithOptions successful!\n");
}
```

## Related Information

- “T02\_createOptionList—Create a TPF Collection Support Option List” on page 990
- “T02\_createSortedBag—Create an Empty Persistent Sorted Bag Collection” on page 1013
- “T02\_createSortedBagTemp—Create a Temporary Sorted Bag Collection” on page 1015.



## T02\_createSortedSet—Create a Persistent Sorted Set Collection

This function creates an empty persistent sorted set collection using the default data definition (DD) and assigns a persistent identifier (PID) to the collection.

A sorted set collection is the same as a sorted bag collection except that the sort fields of added elements must be unique.

### Format

```
#include <c$to2.h>
long T02_createSortedSet ( T02_PID_PTR    pid_ptr,
                          T02_ENV_PTR    env_ptr,
                          const long     *entryLength,
                          const long     *sortFieldLength,
                          const long     *sortFieldDisplacement);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the maximum length of an entry in the collection. The maximum length is 1000 bytes.

#### sortFieldLength

The pointer to the length of the sort field. The maximum length is 256 bytes. The **sortFieldLength** value must be less than or equal to the **entryLength** value.

#### sortFieldDisplacement

The pointer to the displacement in the entry to the start of the sort field. The **sortFieldLength** value plus the **sortFieldDisplacement** value must be less than or equal to the **entryLength** value.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_LOCATOR_LGH
T02_ERROR_LOCATOR_NOT_UNIQUE
```

### Programming Considerations

A commit scope will be created for the T02\_createSortedSet request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createSortedSet request, the scope will be rolled back.

## T02\_createSortedSet

### Examples

The following example creates a persistent sorted set collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;        /* PTR to the T02 environment */
T02_PID      collect;       /* will hold collection's PID */
long         entryLength;   /* PTR to Max Entry Length */
long         sortFieldLength; /* PTR to field length */
long         sortFieldDisplacement; /* PTR to keyFLD */
:
:
if (T02_createSortedSet(&collect,
                        env_ptr,
                        &entryLength,
                        &sortFieldLength,
                        &sortFieldDisplacement) == T02_ERROR)
{
    printf("T02_createSortedSet failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createSortedSet successful!\n");
}
```

### Related Information

- “T02\_createSortedSetTemp—Create a Temporary Sorted Set Collection” on page 1021
- “T02\_createSortedSetWithOptions—Create an Empty Sorted Set Collection with Options” on page 1023.

## TO2\_createSortedSetTemp—Create a Temporary Sorted Set Collection

This function creates a temporary sorted set collection and assigns it a persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is deleted when the ECB exits.

A sorted set collection is the same as a sorted bag collection except that the sort fields of added elements must be unique.

### Format

```
#include <c$to2.h>
long TO2_createSortedSetTemp (   T02_PID_PTR   pid_ptr,
                                T02_ENV_PTR   env_ptr,
                                const long     *entryLength,
                                const long     *sortFieldLength,
                                const long     *sortFieldDisplacement);
```

#### pid\_ptr

The pointer to the location where the PID assigned to the new collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### entryLength

The pointer to the maximum length of an entry in the collection. The maximum length is 1000 bytes.

#### sortFieldLength

The pointer to the length of the sort field. The maximum length is 256 bytes. The **sortFieldLength** value must be less than or equal to the **entryLength** value.

#### sortFieldDisplacement

The pointer to the displacement in the entry to the start of the sort field. The **sortFieldLength** value plus the **sortFieldDisplacement** value must be less than or equal to the **entryLength** value.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_LOCATOR_NOT_UNIQUE
```

### Programming Considerations

- Enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection even though the temporary

## T02\_createSortedSetTemp

collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.

- This function does not use TPF transaction services on behalf of the caller.
- Any references to the PID of the temporary collection after the ECB that created it exits will cause unexpected results.

## Examples

The following example creates a temporary sorted set collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;       /* PTR to the T02 environment */
T02_PID collect;          /* will hold collection's PID */
long entryLength;         /* PTR to Max Entry Length */
long sortFieldLength;     /* PTR to field length */
long sortFieldDisplacement; /* PTR to keyFLD */
:
:
if (T02_createSortedSetTemp(&collect,
                           env_ptr,
                           &entryLength,
                           &sortFieldLength,
                           &sortFieldDisplacement) == T02_ERROR)
{
    printf("T02_createSortedSetTemp failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createSortedSetTemp successful!\n");
}
:
:
T02_deleteCollection(pid_ptr, env_ptr);
```

## Related Information

- “T02\_createSortedSet—Create a Persistent Sorted Set Collection” on page 1019
- “T02\_createSortedSetWithOptions—Create an Empty Sorted Set Collection with Options” on page 1023.

## TO2\_createSortedSetWithOptions—Create an Empty Sorted Set Collection with Options

This function creates an empty sorted set collection using the specified options and assigns a persistent identifier (PID) to the collection.

A sorted set collection is the same as a sorted bag collection except that the sort fields of added elements must be unique.

### Format

```
#include <c$to2.h>
long TO2_createSortedSetWithOptions (TO2_PID_PTR    pid_ptr,
                                     TO2_ENV_PTR    env_ptr,
                                     const TO2_OPTION_PTR optionListPtr,
                                     const long      *entryLength,
                                     const long      *sortFieldLength,
                                     const long      *sortFieldDisplacement);
```

#### **pid\_ptr**

The pointer to the location where the PID assigned to the new collection will be returned.

#### **env\_ptr**

The pointer to the environment as returned by the TO2\_createEnv function.

#### **optionListPtr**

The pointer to a returned options list from a TO2\_createOptionList function call or NULL if no options are required.

#### **entryLength**

The pointer to the maximum length of an entry in the collection. The maximum length is 1000 bytes.

#### **sortFieldLength**

The pointer to the length of the sort field. The maximum length is 256 bytes. The **sortFieldLength** value must be less than or equal to the **entryLength** value.

#### **sortFieldDisplacement**

The pointer to the displacement in the entry to the start of the sort field. The **sortFieldLength** value plus the **sortFieldDisplacement** value must be less than or equal to the **entryLength** value.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_LOCATOR_NOT_UNIQUE
```

## T02\_createSortedSetWithOptions

### Programming Considerations

- A commit scope will be created for the T02\_createSortedSetWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createSortedSetWithOptions request, the scope will be rolled back.
- The T02\_createSortedSetWithOptions request creates either a persistent or temporary collection depending on the data definition (DD).
- Any references to the PID of a temporary collection after the entry control block (ECB) that created it exits will cause unexpected results.

### Examples

The following example creates a persistent sorted set collection.

```
#include <c$to2.h>                                /* Needed for T02 API funcs */
#include <stdio.h>                                /* APIs for standard I/O funcs */

T02_ENV_PTR    env_ptr;                          /* Pointer to the T02 environ. */
T02_PID        collect;                          /* will hold collection's PID */
u_char         ddname[] = "NAME_OF_USER_DD_DEFINITION ";
long           entryLength;                      /* PTR to Max Entry Length */
long           sortFieldLength;                 /* PTR to field length */
long           sortFieldDisplacement;           /* PTR to key field length disp. */
T02_OPTION_PTR optionListPtr;
char           recoupname[] "HOSP0001";

:
/* invoke T02 to create opt lst */
optionListPtr=T02_createOptionList(env_ptr,
                                   T02_OPTION_LIST_CREATE, /* create options */
                                   T02_CREATE_SHADOW,      /* use shadowing */
                                   T02_CREATE_DD,          /* use provided ddname */
                                   T02_CREATE_RECOUP,       /* use provided recoup */
                                   T02_OPTION_LIST_END,     /* end of options */
                                   ddname,                 /* address of ddname */
                                   recoupname);            /* addr of recoup key */

if (optionListPtr == T02_ERROR)
{
    process_error(env_ptr);
}
if (T02_createSortedSetWithOptions(&collect,
                                   env_ptr,
                                   optionListPtr,
                                   &entryLength,
                                   &sortFieldLength,
                                   &sortFieldDisplacement) == T02_ERROR)
{
    printf("T02_createSortedSetWithOptions failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createSortedSetWithOptions successful!\n");
}
```

### Related Information

- “T02\_createOptionList—Create a TPF Collection Support Option List” on page 990
- “T02\_createSortedBag—Create an Empty Persistent Sorted Bag Collection” on page 1013

## **TO2\_createSortedSetWithOptions**

- “TO2\_createSortedBagTemp—Create a Temporary Sorted Bag Collection” on page 1015 .

## **T02\_createSortTemp—Create a Temporary Sorted Collection**

This function has been renamed to T02\_createSortedBagTemp. For more information, see “T02\_createSortedBagTemp—Create a Temporary Sorted Bag Collection” on page 1015.



---

## TO2\_createSortWithOptions—Create an Empty Sorted Collection with Options

This function has been renamed to `TO2_createSortedBagWithOptions`. For more information, see “TO2\_createSortedBagWithOptions—Create an Empty Sorted Bag Collection with Options” on page 1017.

## T02\_definePropertyForPID—Define or Change a Property for a Collection

This function defines or changes a property. The property can only be changed if the new and old types match and if the mode is NORMAL.

### Format

```
#include <c$to2.h>
long T02_definePropertyForPID (const T02_PID_PTR   pid_ptr,
                                T02_ENV_PTR       env_ptr,
                                const char        name[],
                                const enum T02_PROPERTY_TYPE *type,
                                const long        *valueLength,
                                const void        *value);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection for which the property is being defined.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### name

A string that is the name of the property being defined or redefined. The maximum name length is 64 characters.

#### type

A symbol that has one of the following values:

##### TO2\_PROPERTY\_CHAR

Property value is a character. TPFCS assumes a 1-byte length.

##### TO2\_PROPERTY\_DOUBLE

Property value is a double integer (8 bytes). The maximum for a DOUBLE type is  $2^{63} - 1$ . It is stored and displayed as the internal representation of the number. This consists of a sign bit, a 7-bit hexadecimal characteristic, and a 24-bit hexadecimal fraction (significand), which is preceded by an implied decimal point. The number is calculated as:

$$(-1) \times (\text{sign bit}) \times (\text{significand}) \times (\text{Base}^{\text{Exponent}})$$

where: Exponent=characteristic – Bias and the Bias=64 and the Base=16 in an System/370 environment.

##### TO2\_PROPERTY\_LONG

Property value is a long integer (4 bytes).

##### TO2\_PROPERTY\_STRING

Property value is a C string. The string must be delimited by a NULL character. The maximum string length is 256 characters.

##### TO2\_PROPERTY\_STRUCT

Property value is a structure. The valueLength parameter must contain the length of the struct. The maximum struct length is 1000 bytes.

#### valueLength

The length of the value that will be assigned to the defined property.

#### value

The value that will be assigned to the defined property based on the type selected.

## Normal Return

The normal return is a positive value.

## Error Return

An error return is indicated by a zero. When zero is returned, use the `T02_getErrorCode` function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```

T02_ERROR_DATA_LGH
T02_ERROR_DELETED_PID
T02_ERROR_ENV
T02_ERROR_MODE_MISMATCH
T02_ERROR_PARAMETER
T02_ERROR_PID
T02_ERROR_PROPERTY_TYPE
T02_ERROR_ZERO_PID

```

## Programming Considerations

- A commit scope will be created for the `T02_definePropertyForPID` request. If the request is successful, the scope will be committed. If an error occurs while processing the `T02_definePropertyForPID` request, the scope will be rolled back.
- The default mode for properties that are created using this API is **T02\_PROPERTY\_NORMAL**. To specify a different mode, use `T02_definePropertyWithModeForPID`.

## Examples

The following example defines a property attribute to a TPFCS database collection.

```

#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */
T02_PID        keyset;
char propertyName[32] = "X.Property.attribute";
T02_PROPERTY_TYPE propertyType = T02_PROPERTY_CHAR;
char value = 'x';
long valueLength = sizeof(value);

:
if (T02_definePropertyForPID(&keyset,
                             env_ptr,
                             propertyName,
                             &propertyType,
                             &valueLength,
                             &value) == T02_ERROR)
{
    printf("T02_definePropertyForPID failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_definePropertyForPID successful!\n");
:

```

## Related Information

- “`T02_definePropertyWithModeForPID`—Define a Property with a Mode” on page 1031

## **TO2\_definePropertyForPID**

- “TO2\_deleteAllPropertiesFromPID–Delete All Defined Properties” on page 1034
- “TO2\_deletePropertyFromPID–Delete a Property” on page 1038
- “TO2\_getAllPropertyNamesFromPID–Return Property Names” on page 1043
- “TO2\_getPropertyValueFromPID–Retrieve a Property Value” on page 1060
- “TO2\_isPropertyDefinedForPID–Test If a Property Is Already Defined” on page 1068.

## TO2\_definePropertyWithModeForPID—Define a Property with a Mode

This function defines a property with a mode. This function cannot be used to change an existing property value.

### Format

```
#include <c$to2.h>
long TO2_definePropertyWithModeForPID (const T02_PID_PTR  pid_ptr,
                                         T02_ENV_PTR    env_ptr,
                                         const char      name[],
                                         const enum T02_PROPERTY_TYPE *type,
                                         const long       *valueLength,
                                         const void       *value,
                                         const enum T02_PROPERTY_MODE *mode);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection for which the property is being defined.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### name

A string that is the name of the property being defined. The maximum name length is 64 characters.

#### type

A symbol that has one of the following values:

##### TO2\_PROPERTY\_CHAR

Property value is a character. TPFCS assumes a 1-byte length.

##### TO2\_PROPERTY\_DOUBLE

Property value is a double integer (8 bytes). The maximum for a DOUBLE type is  $2^{63} - 1$ . It is stored and displayed as the internal representation of the number. This consists of a sign bit, a 7-bit hexadecimal characteristic, and a 24-bit hexadecimal fraction (significand), which is preceded by an implied decimal point. The number is calculated as:

$$(-1) \times (\text{sign bit}) \times (\text{significand}) \times (\text{Base}^{\text{Exponent}})$$

where: Exponent=characteristic – Bias and the Bias=64 and the Base=16 in an System/370 environment.

##### TO2\_PROPERTY\_LONG

Property value is a long integer. TPFCS assumes a length of 4 bytes.

##### TO2\_PROPERTY\_STRING

Property value is a C string. The string must be delimited by a NULL character. The maximum string length is 256 characters.

##### TO2\_PROPERTY\_STRUCT

Property value is a structure. The valueLength parameter must contain the length of the struct. The maximum struct length is 1000 bytes.

#### valueLength

The length of the value that will be assigned to the defined property.

#### value

The value that will be assigned to the defined property.

#### mode

A character string with one of the following values:

## TO2\_definePropertyWithModeForPID

### TO2\_PROPERTY\_NORMAL

There are no restrictions on the property you can read, change, or delete.

### TO2\_PROPERTY\_NOCHANGE

The property can only be read and deleted. It cannot be changed.

### TO2\_PROPERTY\_NODELETE

The property can be read or changed, but it cannot be deleted.

### TO2\_PROPERTY\_READONLY

The property can only be read. It cannot be changed or deleted. A TO2\_PROPERTY\_READONLY mode is deleted only when the target collection is deleted.

## Normal Return

The normal return is a positive value.

## Error Return

An error return is indicated by a zero. When zero is returned, the T02\_getErrorCode function can be used to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_DELETED_PID
TO2_ERROR_ENV
TO2_ERROR_PARAMETER
TO2_ERROR_PID
TO2_ERROR_PROPERTY_MODE
TO2_ERROR_PROPERTY_TYPE
TO2_ERROR_ZERO_PID
```

## Programming Considerations

A commit scope will be created for the T02\_definePropertyWithModeForPID request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_definePropertyWithModeForPID request, the scope will be rolled back.

## Examples

The following example defines a property attribute with a mode for a TPFCS database collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */
T02_PID        keyset;

T02_PROPERTY_TYPE propertyType = T02_PROPERTY_CHAR;
T02_PROPERTY_MODE propertyMode = T02_PROPERTY_NORMAL;

char propertyName[32] = "X.Property.attribute";
char value = 'x';
long valueLength = sizeof(value);

:
if (T02_definePropertyWithModeForPID(&keyset,
                                     env_ptr,
                                     propertyName,
```

## TO2\_definePropertyWithModeForPID

```
&propertyType,  
&valueLength,  
&value,  
&propertyMode) == T02_ERROR)  
{  
    printf("TO2_definePropertyWithModeForPID failed!\n");  
    process_error(env_ptr);  
}  
else  
    printf("TO2_definePropertyWithModeForPID successful!\n");
```

## Related Information

- “TO2\_definePropertyForPID—Define or Change a Property for a Collection” on page 1028
- “TO2\_deleteAllPropertiesFromPID—Delete All Defined Properties” on page 1034
- “TO2\_deletePropertyFromPID—Delete a Property” on page 1038
- “TO2\_getAllPropertyNamesFromPID—Return Property Names” on page 1043
- “TO2\_getPropertyValueFromPID—Retrieve a Property Value” on page 1060
- “TO2\_isPropertyDefinedForPID—Test If a Property Is Already Defined” on page 1068.

### TO2\_deleteAllPropertiesFromPID—Delete All Defined Properties

This function deletes all the defined properties that can be deleted. Properties with either TO2\_PROPERTY\_NODELETE or TO2\_PROPERTY\_READONLY modes will not be deleted.

#### Format

```
#include <c$to2.h>
long TO2_deleteAllPropertiesFromPID (const T02_PID_PTR  pid_ptr,
                                     T02_ENV_PTR  env_ptr);
```

##### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that has properties that will be deleted.

##### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### Normal Return

The normal return is a positive value. A positive value will be returned even if all properties are not deleted (for example, some may be marked read-only).

#### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DELETED_PID
TO2_ERROR_ENV
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

A commit scope will be created for the T02\_deleteAllPropertiesFromPID request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_deleteAllPropertiesFromPID request, the scope will be rolled back.

### Examples

The following example deletes all property attributes from a TPFCS database collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;            /* Pointer to T02 Environment */
T02_PID      keyset;
:
:
if (T02_deleteAllPropertiesFromPID(&keyset,
                                  env_ptr) == T02_ERROR)
{
    printf("T02_deleteAllPropertiesFromPID failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_deleteAllPropertiesFromPID successful!\n");
```



## **Related Information**

- “TO2\_definePropertyForPID—Define or Change a Property for a Collection” on page 1028
- “TO2\_definePropertyWithModeForPID—Define a Property with a Mode” on page 1031
- “TO2\_getAllPropertyNamesFromPID—Return Property Names” on page 1043
- “TO2\_deletePropertyFromPID—Delete a Property” on page 1038
- “TO2\_getPropertyValueFromPID—Retrieve a Property Value” on page 1060
- “TO2\_isPropertyDefinedForPID—Test If a Property Is Already Defined” on page 1068.

## T02\_deleteCollection–Delete a Collection

This function deletes a collection and is identical to the T02\_deletePID function. Any backing store assigned to the collection will be released.

After you enter the T02\_deleteCollection function, the collection is either marked for deletion or actually deleted from the database and cannot be accessed by other TPFCS functions. For persistent short-term collections and temporary collections, the delete always takes place immediately. For persistent long-term collections, the delete is controlled by the data store characteristics set with the ZOODB command. A persistent long-term collection marked for deletion can be reclaimed by entering the T02\_reclaimPID function in the time period between the T02\_deleteCollection request and the time the actual deletion occurs (48 hours).

### Format

```
#include <c$to2.h>
long T02_deleteCollection (const T02_PID_PTR   pid_ptr,
                           T02_ENV_PTR       env_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that will be deleted.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_NOT_INIT
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

### Programming Considerations

A commit scope will be created for the T02\_deleteCollection request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_deleteCollection request, the scope will be rolled back.

### Examples

The following example deletes a collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;    /* Pointer to T02 Environment */
T02_PID       bag;
T02_ERR_CODE  err_code;

void test_delete_collections(void)
```

```
{
  :
  if (TO2_deleteCollection(&bag,
                          env_ptr) == TO2_ERROR)
  {
    printf("deleteCollection for Bag failed!\n");
    process_error(env_ptr);
  }
  else
  {
    printf("deleteCollection for Bag successful!\n");
  }
  return;
}
```

## Related Information

- “TO2\_deletePID–Delete a Persistent Identifier and Its Backing Store” on page 1352
- “TO2\_makeEmpty–Delete All Elements from a Collection” on page 1072
- “TO2\_reclaimPID–Reclaim a PID” on page 1322.

### TO2\_deletePropertyFromPID–Delete a Property

This function deletes the specified named property.

#### Format

```
#include <c$to2.h>
long TO2_deletePropertyFromPID (const T02_PID_PTR    pid_ptr,
                                T02_ENV_PTR          env_ptr,
                                const char           name[]);
```

##### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection for which property is being deleted.

##### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

##### name

A string that is the name of the property being deleted. The maximum length is 64 characters.

#### Normal Return

The normal return is a positive value. A positive value will not be returned if all properties are not deleted.

#### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ACCESS_MISMATCH
TO2_ERROR_DATA_LGH
TO2_ERROR_DELETED_PID
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

A commit scope will be created for the T02\_deletePropertyFromPID request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_deletePropertyFromPID request, the scope will be rolled back.

### Examples

The following example deletes a specific property attribute from a TPFCS database collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;     /* Pointer to T02 Environment */
T02_PID        keyset;

char propertyname[32] = "X.Property.attribute";
:
:
if (T02_deletePropertyFromPID(&keyset,
                             env_ptr,
```

## TO2\_deletePropertyFromPID

```
propertyname) == T02_ERROR)
{
    printf("TO2_deletePropertyFromPID failed!\n");
    process_error(env_ptr);
}
else
    printf("TO2_deletePropertyFromPID successful!\n");
```

### Related Information

- “TO2\_definePropertyForPID—Define or Change a Property for a Collection” on page 1028
- “TO2\_definePropertyWithModeForPID—Define a Property with a Mode” on page 1031
- “TO2\_deleteAllPropertiesFromPID—Delete All Defined Properties” on page 1034
- “TO2\_getAllPropertyNamesFromPID—Return Property Names” on page 1043
- “TO2\_getPropertyValueFromPID—Retrieve a Property Value” on page 1060
- “TO2\_isPropertyDefinedForPID—Test If a Property Is Already Defined” on page 1068.

## T02\_deleteRecoupIndex–Delete a Recoup Index

This function deletes a TPF collection support (TPFCS) recoup index.

### Format

```
#include <c$to2.h>
long T02_deleteRecoupIndex (      T02_ENV_PTR  env_ptr,
                                const void    *indexName);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### indexName

The pointer to the 8-byte recoup index name for the recoup index that will be deleted.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

T02\_ERROR\_ENV

## Programming Considerations

None.

## Examples

The following example deletes a recoup index.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */
T02_ENV_PTR    env_ptr;          /* Pointer to T02 environment */

u_char        indexName[]="INDEX003"; /* index identifier */
:
:
if (T02_deleteRecoupIndex(env_ptr,
                          indexName) == T02_ERROR)
{
    printf ("T02_deleteRecoupIndex failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_deleteRecoupIndex successful!\n");
```

## Related Information

- “T02\_createRecoupIndex–Create a Recoup Index” on page 998
- “T02\_removeRecoupIndexFromPID–Remove a PID to Index Association” on page 1086.

## T02\_deleteRecoupIndexEntry—Delete an Entry from an Index

This function is used to delete an entry from a TPF collection support recoup index.

### Format

```
#include <c$to2.h>
long T02_deleteRecoupIndexEntry (      T02_ENV_PTR  env_ptr,
                                       const void    *indexName,
                                       const void    *entryToken);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### indexName

The pointer to the 8-byte recoup index name for the recoup index that will be deleted.

#### entryToken

A user-defined token that was used to identify the index entry when it was created.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

T02\_ERROR\_ENV

## Programming Considerations

Delete any references requiring recoup that are affected by this change before the next occurrence of recoup.

### Examples

The following example deletes an entry from an index.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 environment */
char           token[8]

u_char         indexName="INDEX002"; /* index identifier */
:
:
if (T02_deleteRecoupIndexEntry(env_ptr,
                               indexName,
                               token) == T02_ERROR)
{
    printf ("T02_deleteRecoupIndexEntry failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_deleteRecoupIndexEntry successful!\n");
```

## **TO2\_deleteRecoupIndexEntry**

### **Related Information**

- “TO2\_addRecoupIndexEntry–Add an Entry to a Recoup Index” on page 893
- “TO2\_deleteRecoupIndex–Delete a Recoup Index” on page 1040
- “TO2\_removeRecoupIndexFromPID–Remove a PID to Index Association” on page 1086.



## T02\_getAllPropertyNamesFromPID—Return Property Names

This function returns a collection containing all the defined property names.

### Format

```
#include <c$to2.h>
long T02_getAllPropertyNamesFromPID (const T02_PID_PTR  pid_ptr,
                                         T02_ENV_PTR  env_ptr,
                                         T02_PID_PTR  return_pid_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that will be interrogated.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### return\_pid\_ptr

The pointer to where to store the PID of a temporary collection that contains all the defined property names.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DELETED_PID
TO2_ERROR_ENV
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example retrieves a list of property names for a TPFCS database collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* Pointer to T02 Environment */
T02_PID      keyset;

T02_PID      property_list; /* PID for collection object */
:
:
if (T02_getAllPropertyNamesFromPID(&keyset,
                                   env_ptr,
                                   &property_list) == T02_ERROR)
{
    printf("T02_getAllPropertyNamesFromPID failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_getAllPropertyNamesFromPID successful!\n");
```

## TO2\_getAllPropertyNamesFromPID

### Related Information

- “TO2\_deleteAllPropertiesFromPID–Delete All Defined Properties” on page 1034
- “TO2\_deletePropertyFromPID–Delete a Property” on page 1038
- “TO2\_definePropertyForPID–Define or Change a Property for a Collection” on page 1028
- “TO2\_definePropertyWithModeForPID–Define a Property with a Mode” on page 1031
- “TO2\_getPropertyValueFromPID–Retrieve a Property Value” on page 1060
- “TO2\_isPropertyDefinedForPID–Test If a Property Is Already Defined” on page 1068.

## TO2\_getBLOB—Retrieve the Contents of a BLOB

This function allows an application program to retrieve the contents of a binary large object (BLOB). TPF collection support (TPFCS) will allocate a malloc storage buffer and read the contents of the BLOB into the storage area. A pointer to the storage area will be returned to the caller with the current sequence counter value of the BLOB. The size that TPFCS reads is only limited by the amount of malloc storage available for allocation.

**Note:** This function only supports BLOB collections.

### Format

```
#include <c$to2.h>
void * TO2_getBLOB (const T02_PID_PTR  pid_ptr,
                      T02_ENV_PTR    env_ptr,
                      long            *dataLength,
                      long            *sequenceCounter);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment collection as returned by the T02\_createEnv function.

#### dataLength

The pointer to a field that TPFCS will overlay with the length of the read data.

#### sequenceCounter

The pointer to a field that TPFCS will overlay with the current sequence counter value.

### Normal Return

The normal return is a pointer to a buffer containing the requested data.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_STORAGE
TO2_ERROR_ZERO_PID
```

**Note:** The update sequence counter value will also be returned for the TO2\_ERROR\_EMPTY error.

### Programming Considerations

- It is the responsibility of the caller to free the returned buffer once the caller has stopped using the buffer.
- This function does not use TPF transaction services on behalf of the caller.

## TO2\_getBLOB

### Examples

The following example retrieves the contents of a BLOB.

```
#include <c$to2.h>           /* T02 API function prototypes */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;     /* Pointer to T02 Environment */
T02_PID        blob;
long           seqCtr;      /* field to hold sequence counter */
long           dataLgh;     /* field to hold length of BLOB */
char           *bufPtr;

:
:
if ((bufPtr = (T02_getBLOB(&blob,
                        env_ptr,
                        &dataLgh,
                        &seqCtr)) == T02_ERROR)
{
    printf("T02_getBLOB failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_getBLOB was successful!\n");
}
:
:
free(bufPtr);
```

### Related Information

- “TO2\_atRBA—Retrieve Data from a BLOB” on page 915
- “TO2\_atRBAPut—Store Data in a BLOB” on page 917
- “TO2\_getBLOBwithBuffer—Retrieve Contents of a BLOB Using a Passed Buffer” on page 1047
- “TO2\_removeRBA—Remove an Area from a BLOB” on page 1084
- “TO2\_replaceBLOB—Replace the Contents of a BLOB with New Data” on page 1092
- “TO2\_size—Return the Size of the Collection” on page 1109
- “TO2\_writeNewBLOB—Create a New BLOB and Add the Passed Data” on page 1111.

## TO2\_getBLOBwithBuffer–Retrieve Contents of a BLOB Using a Passed Buffer

This function allows an application program to retrieve the current contents of a binary large object (BLOB). TPF collection support (TPFCS) will read the contents of the BLOB into the storage area passed as a parameter along with the current sequence counter value of the BLOB. The size that is read is determined by the size of your buffer or the size of the BLOB, whichever is smaller.

**Note:** This function only supports BLOB collections.

### Format

```
#include <c$to2.h>
void * TO2_getBLOBwithBuffer (const T02_PID_PTR    pid_ptr,
                                T02_ENV_PTR      env_ptr,
                                void             *buffer,
                                long             *bufferLength,
                                long             *sequenceCounter);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment collection as returned by the T02\_createEnv function.

#### buffer

The pointer to the buffer to use to read the contents of the BLOB.

#### bufferLength

The pointer to a field that contains the length of your buffer. TPFCS will overlay the field with the actual length of the data for the BLOB. The application program must compare the returned length of the BLOB data with the length of the buffer to determine if the data for the BLOB was completely read.

#### sequenceCounter

The pointer to a field that TPFCS will overlay with the current sequence counter value.

### Normal Return

The normal return is a pointer to the passed buffer.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_EMPTY
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

**Note:** The update sequence counter value will also be returned for the TO2\_ERROR\_EMPTY error.

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example retrieves the contents of a BLOB.

```
#include <c$to2.h>           /* T02 API function prototypes */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;      /* Pointer to T02 Environment */
T02_PID        blob;
long           seqCtr;       /* field to hold sequence counter */
long           bufferLength=400; /* field to hold length of BLOB */
long           userBufLgh=400; /* user buffer length */
char           buffer[400];

:
:
if (T02_getBLOBwithBuffer(&blob,
                        env_ptr,
                        buffer,
                        &bufferLength,
                        &seqCtr) == T02_ERROR)
{
    printf("T02_getBLOBwithBuffer failed!\n");
    process_error(env_ptr);
}
else if (bufferLength < userBufLgh)
    printf("BLOB larger than passed buffer\n");
else
    printf("T02_getBLOBwithBuffer was successful!\n");
```

### Related Information

- “TO2\_atRBA—Retrieve Data from a BLOB” on page 915
- “TO2\_atRBAPut—Store Data in a BLOB” on page 917
- “TO2\_getBLOB—Retrieve the Contents of a BLOB” on page 1045
- “TO2\_removeRBA—Remove an Area from a BLOB” on page 1084
- “TO2\_replaceBLOB—Replace the Contents of a BLOB with New Data” on page 1092
- “TO2\_size—Return the Size of the Collection” on page 1109
- “TO2\_writeNewBLOB—Create a New BLOB and Add the Passed Data” on page 1111.

## TO2\_getCollectionAccessMode — Retrieve the Access Mode of a Collection

This function returns the current access mode of a collection.

### Format

```
#include <c$to2.h>
T02_COLLECT_MODE T02_getCollectionAccessMode (const T02_PID_PTR pid_ptr;
                                              T02_ENV_PTR env_ptr);
```

#### pid\_ptr

A pointer to the persistent identifier (PID) of the collection whose access mode value will be retrieved.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is an enumerated value of the type T02\_COLLECT\_MODE. For more information, see “T02\_setCollectionAccessMode — Set the Access Mode of a Collection” on page 1101.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DELETED_PID
T02_ERROR_NOT_INIT
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

### Programming Considerations

TPFCS defaults the access mode of a collection to T02\_COLLECT\_MODE\_NORMAL. To change the access mode of a collection, use the T02\_setCollectionAccessMode function.

### Examples

The following example tests to see if a collection can be deleted and returns a TRUE (1) value if it can be deleted or a FALSE (0) value if it cannot be deleted.

```
#include <c$to2.h>          /* Needed for T02 API functions */

long  deleteAllowed(const T02_PID_PTR  pid_ptr,
                      T02_ENV_PTR     env_ptr);

:
/*
/*  retrieve the collections access mode and test it for
/*  deletion allowed.
/*
long  deleteAllowed(const T02_PID_PTR  pid_ptr,
                      T02_ENV_PTR     env_ptr)
{
    T02_COLLECT_MODE  accessMode,
```

## TO2\_getCollectionAccessMode

```
if ((accessMode = TO2_getCollectionAccessMode(pid_ptr,
                                              env_ptr)) == TO2_ERROR)
{
    process_error(env_ptr);
    return -1;
}
else if ((accessMode == TO2_COLLECT_MODE_NODELETE) ||
        (accessMode == TO2_COLLECT_MODE_READONLY))
    return 0; /* collection cannot be deleted. */
else return 1; /* collection can be deleted. */
}
```

## Related Information

- “TO2\_readOnly—Get the Read-Only Attribute of the Collection” on page 1076
- “TO2\_setCollectionAccessMode — Set the Access Mode of a Collection” on page 1101.



## T02\_getCollectionKeys—Get the Primary Key Values of the Collection

This function returns a temporary sequence collection containing the primary key values for all elements in the collection.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_getCollectionKeys(T02_PID_PTR pid_ptr,
                          T02_ENV_PTR env_ptr,
                          T02_PID_PTR retnPid_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection whose keys will be retrieved.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### retnPid\_ptr

The pointer to where the PID for the temporary sequence collection will be stored.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
```

### Programming Considerations

- You must enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the entry control block (ECB) exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- The returned sequence collection will contain the primary keys for a collection, regardless of whether alternate key paths are defined for the collection.
- This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example creates a temporary sequence collection containing the keys of all the elements in the source collection.

```
#include <c$to2.h>          /* needed for TPFCS API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_PID   keyedCol;         /* PID of source collection */
T02_ENV_PTR env_ptr;       /* pointer to the TPFCS environment */
```

## TO2\_getCollectionKeys

```
TO2_PID      keys;          /* PID of sequence collection of keys */
:
if (TO2_getCollectionKeys(&keyedCol, env_ptr, &keys) == TO2_ERROR)
{
    printf("TO2_getCollectionKeys failed!\n");
    process_error(env_ptr);
    return;
}
else
{
    printf("TO2_getCollectionKeys successful!\n");
}
:
TO2_deleteCollection(&keys, env_ptr);
```

## Related Information

- “TO2\_deleteCollection—Delete a Collection” on page 1036
- “TO2\_getCurrentKey—Retrieve the Current Key” on page 1141
- “TO2\_key—Return Current Key Value” on page 1149.

## TO2\_getCollectionType—Get the Type Value of the Collection

This function retrieves a pointer to a string that identifies the type of the collection.

### Format

```
#include <c$to2.h>
char * T02_getCollectionType (const T02_PID_PTR    pid_ptr,
                               T02_ENV_PTR      env_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) of the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a pointer to a C string, TYPE is the type of the collection and CODE is the string returned:

TYPE	CODE
array	ARRAY
bag	BAG
BLOB	BYTEARRAY
key bag	KEYBAG
key set	KEYSET
key sorted bag	KEYSORTEDBAG
key sorted set	KEYSORTEDSET
keyed log	KEYEDLOG
log	LOG
sequence	SEQUENCE
set	SET
sorted bag	SORTEDBAG
sorted set	SORTEDSET

### Error Return

An error return is indicated by a NULL pointer. When a NULL pointer is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

## T02\_getCollectionType

### Examples

The following example retrieves a string-identifier type for the specified TPFCS collection.

```
#include <c$to2.h>           /* T02 API function prototypes */
#include <string.h>          /* strcmp() */
#include <stdio.h>           /* APIs for standard I/O functions */

void processVariousCollectionTypes (T02_ENV_PTR env_ptr,
                                   T02_PID_PTR pid_ptr)
{
    char *objecttype = T02_getCollectionType(pid_ptr, env_ptr);

    if (objecttype == NULL)
    {

        printf("T02_getCollectionType failed!\n");
        process_error(env_ptr);
    }
    else
    {
        if (strcmp(objecttype, "ARRAY") == 0)
        {
            /* Processing for ARRAY type collection */
        }
        else if (strcmp(objecttype, "BAG") == 0)
        {
            /* Processing for BAG type collection */
        }
        else if (strcmp(objecttype, "BYTEARRAY") == 0)
        {
            /* Processing for BLOB type collection */
        }
        :
        printf("T02_getCollectionType successful!\n");
    }
}
```

### Related Information

“T02\_isCollection—Test If PID Is for a Collection” on page 1066.

## TO2\_getDRprotect — Retrieve Dirty-Reader Protection Status

This function returns the dirty-reader protection status of the collection. Dirty-reader protection determines whether TPFCS will use the FILNC macro or the FILEC macro to file records. If dirty-reader protection is on (TO2\_IS\_TRUE), TPFCS will use the FILNC macro to file records. TPF transaction services maintains a copy of each record filed using a FILNC macro. Maintaining a copy of each record filed can cause the size of TPF transaction services to be enlarged. It can also cause TPF transaction services to either exceed the maximum allowed size or to actually run out of TPF transaction services buffers. So, dirty-reader protection should **only** be set on a collection that will be accessed by dirty readers.

### Format

```
#include <c$to2.h>
BOOL TO2_getDRprotect (const T02_PID_PTR  pid_ptr,
                        T02_ENV_PTR  env_ptr);
```

#### pid\_ptr

A pointer to the persistent identifier (PID) of the collection whose dirty-reader protection state will be retrieved.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

#### TO2\_IS\_TRUE

Dirty-reader protection is active for the collection.

#### TO2\_IS\_FALSE

Dirty-reader protection is not active for the collection.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. This function must also be used with a TO2\_IS\_FALSE return code to distinguish this return from an error return indication. If the error code returned by T02\_getErrorCode is TO2\_IS\_FALSE (0), this is the actual return code by the function. Otherwise, an error is indicated. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DELETED_PID
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

To avoid running out of commit scope buffers or exceeding maximum allowed size, consider the following:

- Do not use TPF transaction services. You will need to decide if this is an option.
- Use TPF transaction services, but keep the number of updates low so that the number of records updated in the commit scope will be small. This may be a better option, but it is hard to predict the number of records that will be updated on any specific call to TPFCS.

## TO2\_getDRprotect

- Do not use dirty readers and, therefore, dirty-reader protection is not needed. This is the TPFCS default mode. Whenever a collection is created, it is created with dirty-reader protection turned off. Therefore all files are filed using FILEC instead of FILNC.
- If dirty readers are needed, then enable dirty-reader protection only on those collections that will actually be read without locks. Do this by issuing the TO2\_setDRprotect function after the collection has been created.
- Once dirty-reader protection has been turned on, it cannot be turned off.

## Examples

The following example retrieves the dirty-reader protection status of the provided collection.

```
#include <c$to2.h>                /* Needed for TO2 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

TO2_PID          pid_ptr;
TO2_ENV_PTR      env_ptr;
TO2_ERR_CODE     err_code;      /* TO2 error code value */
:
:
/*****
/*   retrieve the dirty-reader protect state from the collection   */
/*   and return it to the caller.                                   */
*****/

if (TO2_getDRprotect(pid_ptr, env_ptr)) == TO2_IS_FALSE)
{
    err_code = TO2_getErrorCode(env_ptr);
    if (err_code != 0)
    {
        printf("TO2_getDRprotect failed!\n");
        process_error(env_ptr);
    }
    else
        printf("Dirty-reader protection is not active for the collection!\n");
}
else
    printf("Dirty-reader protection is active for the collection!\n") ;
```

## Related Information

“TO2\_setDRprotect — Set Dirty-Reader Protection On” on page 1103.

## T02\_getMaxDataLength—Retrieve the Maximum Data Length Value

This function returns the maximum data length value that was set when you created the collection.

### Format

```
#include <c$to2.h>
long T02_getMaxDataLength (const T02_PID_PTR pid_ptr,
                             T02_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that will be interrogated.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The maximum data length will be returned as the return value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example prints the maximum data length.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;      /* PTR to the T02 environment */
T02_PID_PTR collect;      /* PTR to PID return field */
long max;                /* Maximum data length */
:
:
if (((max = T02_getMaxDataLength(&collect, env_ptr)) == T02_ERROR)
{
    printf("T02_getMaxDataKeyLength failed!\n");
    process_error(env_ptr);
}
else
{
    printf("Maximum data length is %d.\n", max);
}
```

### Related Information

- “T02\_getMaxKeyLength—Retrieve the Maximum Key Length Value” on page 1058
- “T02\_getSortFieldValues—Retrieve the Sort Field Values” on page 1062.

---

### TO2\_getMaxKeyLength—Retrieve the Maximum Key Length Value

This function returns the maximum key length value that was set when you created the collection.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

#### Format

```
#include <c$to2.h>
long TO2_getMaxKeyLength (const TO2_PID_PTR   pid_ptr,
                           TO2_ENV_PTR      env_ptr);
```

##### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that will be interrogated.

##### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### Normal Return

The maximum key length will be returned as the return value.

#### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

#### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

#### Examples

The following example prints the maximum key length.

```
#include <c$to2.h>          /* Needed for TO2 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

TO2_ENV_PTR   env_ptr;     /* PTR to the TO2 environment */
TO2_PID       collect;     /* PID return field */
long          max;         /* Maximum key length */
:
:
if (((max = TO2_getMaxKeyLength(&collect, env_ptr)) == TO2_ERROR)
{
    printf("TO2_getMaxKeyLength failed!\n");
    process_error(env_ptr);
}
else
{
    printf("Maximum key length is %d.\n", max);
}
```



**Related Information**

- “TO2\_getMaxDataLength–Retrieve the Maximum Data Length Value” on page 1057
- “TO2\_getSortFieldValues–Retrieve the Sort Field Values” on page 1062.

# T02\_getPropertyValueFromPID—Retrieve a Property Value

This function retrieves the current value of the named property.

## Format

```
#include <c$to2.h>
long T02_getPropertyValueFromPID(const T02_PID_PTR    pid_ptr,
                                  T02_ENV_PTR        env_ptr,
                                  const char          name[],
                                  enum T02_PROPERTY_TYPE *type,
                                  long                *valueLength,
                                  void               *value,
                                  enum T02_PROPERTY_MODE *mode);
```

### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that owns the property.

### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### name

A string that is the name of the property being retrieved.

### type

The pointer to an area where the type enumeration of the property can be returned.

### valueLength

The length of the field where the value of the property will be returned. This field will be overlaid with the actual length of the value of the property.

### value

The pointer to the field where the value of the property will be returned.

### mode

The pointer to an area where the mode enumeration of the property can be returned.

## Normal Return

The normal return is a positive value.

## Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_DELETED_PID
T02_ERROR_ENV
T02_ERROR_LOCATOR_NOT_FOUND
T02_ERROR_PARAMETER
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example retrieves the value of a named property attribute from a TPFCS database collection.

```
#include <c$to2.h>           /* Needed for TO2 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

TO2_ENV_PTR    env_ptr;     /* Pointer to TO2 Environment */
TO2_PID        keyset;
TO2_PROPERTY_TYPE propertyType;
TO2_PROPERTY_MODE propertyMode;

char propertyName[32] = "X.Property.attribute";
char value;
long valueLength = sizeof(value);
:
if (TO2_getPropertyValueFromPID(&keyset,
                                env_ptr,
                                propertyName,
                                &propertyType,
                                &valueLength,
                                &value,
                                &propertyMode) == TO2_ERROR)
{
    printf("TO2_getPropertyValueFromPID failed!\n");
    process_error(env_ptr);
}
else
    printf("TO2_getPropertyValueFromPID successful!\n");
```

## Related Information

- “TO2\_deleteAllPropertiesFromPID–Delete All Defined Properties” on page 1034
- “TO2\_deletePropertyFromPID–Delete a Property” on page 1038
- “TO2\_definePropertyForPID–Define or Change a Property for a Collection” on page 1028
- “TO2\_definePropertyWithModeForPID–Define a Property with a Mode” on page 1031
- “TO2\_getAllPropertyNamesFromPID–Return Property Names” on page 1043
- “TO2\_isPropertyDefinedForPID–Test If a Property Is Already Defined” on page 1068.

## T02\_getSortFieldValues—Retrieve the Sort Field Values

This function retrieves the sort field displacement and length that was set when the sorted collection was created.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_getSortFieldValues (const T02_PID_PTR    pid_ptr,
                                T02_ENV_PTR      env_ptr,
                                long              *sortFieldLength,
                                long              *sortFieldDisplacement);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that will be interrogated.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### sortFieldLength

The pointer to the field that will hold the returned sort field length value.

#### sortFieldDisplacement

The pointer to the field that will hold the returned sort field displacement value.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example prints sort field values.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* PTR to the T02 environment */
T02_PID        collect;    /* PTR to PID return field */
long           sortlen;    /* PTR to return location for
                           sort field length value */
long           sortdsp;    /* PTR to return location for
                           sort field displacement value */
:
:
if (T02_getSortFieldValues(&collect,
```

```
env_ptr,  
&sortlen,  
&sortdsp) == T02_ERROR)  
{  
    printf("TO2_getSortFieldValues failed!\n");  
    process_error(env_ptr);  
}  
else  
{  
    printf("Sort field length is %d and displacement is %d.\n", sortlen, sortdsp);  
}
```

## Related Information

- “TO2\_getKeyPathAttributes–Retrieve the Attributes of Key Paths” on page 1281
- “TO2\_getMaxDataLength–Retrieve the Maximum Data Length Value” on page 1057
- “TO2\_getMaxKeyLength–Retrieve the Maximum Key Length Value” on page 1058.

## TO2\_includes–Search Collection for Element Equal to the Argument

This function causes the collection to be searched for an element equal to the argument and returns a TO2\_IS\_TRUE or TO2\_IS\_FALSE indication.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
BOOL TO2_includes (const T02_PID_PTR   pid_ptr,
                   T02_ENV_PTR       env_ptr,
                   const void        *value,
                   const long        *valueLength);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### value

The pointer to the value to test.

#### valueLength

The pointer to the length of the value to test.

### Normal Return

#### TO2\_IS\_TRUE

The collection contains the argument.

#### TO2\_IS\_FALSE

The collection does not contain the argument.

### Error Return

An error return is indicated by a zero return value and a nonzero error code. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For the TO2\_IS\_FALSE result, check the TPFCS error code by entering T02\_getErrorCode to distinguish this result from an error return indication. If the error code is zero, the Boolean result is false. Otherwise, an error is indicated and you can retrieve the error text by entering T02\_getErrorText. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example prints data values if they are present in a collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */
```

## TO2\_includes

```
TO2_ENV_PTR    env_ptr;      /* Pointer to the T02 environment    */
TO2_PID        bag;
char           bagdata[] = "Bag:Collection:Element:.Data";
long           bagdatalength; /* length */

if (TO2_includes(&bag,
                env_ptr,
                &bagdata,
                &bagdatalength) == T02_IS_FALSE)
{
    err_code = T02_getErrorCode(env_ptr);
    if ((err_code != 0))
    {
        printf("T02_includes failed!\n");
        process_error(env_ptr);
    }
    else
    {
        printf("Bag does not include value.\n");
    }
}
else
    printf("Bag includes value.\n");
```

## Related Information

- “TO2\_add—Add an Element to a Collection” on page 883
- “TO2\_removeValue—Remove Value” on page 1088.

## T02\_isCollection—Test If PID Is for a Collection

This function determines if a persistent identifier (PID) is for a collection.

### Format

```
#include <c$to2.h>
BOOL T02_isCollection (const T02_PID_PTR   pid_ptr,
                        T02_ENV_PTR      env_ptr);
```

#### pid\_ptr

The pointer to the PID of the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

#### TO2\_IS\_FALSE

PID is not a collection.

#### TO2\_IS\_TRUE

PID is a collection.

### Error Return

An error return is indicated by a zero return value and a nonzero error code. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- This function does not use TPF transaction services on behalf of the caller.
- For the TO2\_IS\_FALSE result, check the TPF collection support(TPFCS) error code by entering T02\_getErrorCode to distinguish this result from an error return indication. If the error code is zero, the Boolean result is false. Otherwise, an error is indicated and you can retrieve the error text by entering T02\_getErrorText.

### Examples

The following example determines if a specified PID represents a TPFCS collection.

```
#include <c$to2.h>                /* T02 API function prototypes */
#include <stdio.h>                /* APIs for standard I/O functions */
T02_ENV_PTR   env_ptr;          /* Pointer to T02 Environment */
T02_PID       blob;

T02_ERR_CODE  err_code;         /* T02 error code value */
:
:
if (T02_isCollection(&blob,
                    env_ptr) == TO2_IS_FALSE)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != 0)
    {
        printf("T02_isCollection failed!\n");
    }
}
```



```
        process_error(env_ptr);
    }
    else
    {
        printf("Object IS NOT a TPFCS Collection.\n");
    }
}
else
{
    printf("Object IS a TPFCS Collection.\n");
}
```

## Related Information

“TO2\_getCollectionType—Get the Type Value of the Collection” on page 1053.

## T02\_isPropertyDefinedForPID–Test If a Property Is Already Defined

This function determines if the named property is already defined.

### Format

```
#include <c$to2.h>
BOOL T02_isPropertyDefinedForPID (const T02_PID_PTR  pid_ptr,
                                   T02_ENV_PTR      env_ptr,
                                   const char        name[]);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection for which the property is being defined.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### name

A string that is the name of the property being tested. The maximum name length is 64 characters.

### Normal Return

#### TO2\_IS\_FALSE

The property name is not defined.

#### TO2\_IS\_TRUE

The property name is defined.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. This function must also be used with a TO2\_IS\_FALSE return code to distinguish this return from an error return indication. If the error code returned by T02\_getErrorCode is TO2\_IS\_FALSE (0), this is the actual return code by the function. Otherwise, an error is indicated. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_DELETED_PID
TO2_ERROR_ENV
TO2_ERROR_PARAMETER
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example determines if the specified property attribute is currently defined for a TPFCS database collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */
T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */
T02_PID        keyset;

char propertyname[32] = "X.Property.attribute";
```

```

:
if (TO2_isPropertyDefinedForPID(&keyset,
                                env_ptr,
                                propertyname) == TO2_IS_FALSE)
{
    printf("TO2_isPropertyDefinedForPID failed!\n");
    process_error(env_ptr);
}
else
    printf("TO2_isPropertyDefinedForPID successful!\n");

```

## Related Information

- “TO2\_deleteAllPropertiesFromPID—Delete All Defined Properties” on page 1034
- “TO2\_deletePropertyFromPID—Delete a Property” on page 1038
- “TO2\_definePropertyForPID—Define or Change a Property for a Collection” on page 1028
- “TO2\_getAllPropertyNamesFromPID—Return Property Names” on page 1043
- “TO2\_definePropertyWithModeForPID—Define a Property with a Mode” on page 1031
- “TO2\_getPropertyValueFromPID—Retrieve a Property Value” on page 1060.

## T02\_isTemp—Determine If Collection Is Temporary or Persistent

This function determines if the specified persistent identifier (PID) represents a temporary or persistent collection.

### Format

```
#include <c$to2.h>
BOOL T02_isTemp (const T02_PID_PTR pid_ptr,
                  T02_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to the PID assigned to the collection to test.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

A value of T02\_IS\_TRUE is returned if the collection is temporary. A value of T02\_IS\_FALSE is returned if the collection is persistent.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

```
T02_ERROR_ENV
T02_ERROR_PID
```

### Programming Considerations

- For the T02\_IS\_FALSE result, check the TPF collection support (TPFCS) error code by calling the T02\_getErrorCode function to distinguish this result from an error return indication. If the error code is zero, the Boolean result is false. Otherwise, an error is indicated and you can retrieve the error text by calling the T02\_getErrorText function.
- This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example determines if the specified collection is temporary or persistent.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */
T02_ENV_PTR env_ptr;        /* PTR to the T02 environment */
T02_PID_PTR collect;        /* PTR to PID return field */
T02_ERR_CODE err_code;
:
:
if ((T02_isTemp(&collect, env_ptr)) == T02_IS_FALSE)
{
    err_code=T02_getErrorCode(env_ptr);
    if (err_code != T02_IS_FALSE)
    {
        printf (T02_isTemp failed!\n")
        process_error (env_ptr);
    }
}
else
{
    printf("This is a persistent collection! \n");
}
```

```
}  
else  
{  
    printf("This is a temporary collection! \n");  
}
```

## Related Information

None.

## T02\_makeEmpty–Delete All Elements from a Collection

This function reinitializes a collection and discards any elements in the collection.

### Format

```
#include <c$to2.h>
long T02_makeEmpty (const T02_PID_PTR   pid_ptr,
                    T02_ENV_PTR   env_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that will be deleted.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

A commit scope will be created for the T02\_makeEmpty request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_makeEmpty request, the scope will be rolled back.

## Examples

The following example deletes all the elements from a collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */
T02_ENV_PTR   env_ptr;      /* PTR to the T02 environment */
T02_PID_PTR   collect;      /* PTR to PID return field */
:
:
if (T02_makeEmpty(&collect,
                 env_ptr) == T02_ERROR)
{
    printf("T02_makeEmpty failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_makeEmpty successful!\n");
}
```

## Related Information

- “T02\_add–Add an Element to a Collection” on page 883
- “T02\_createBag–Create a Persistent Bag Collection” on page 939

## **TO2\_makeEmpty**

- “TO2\_createSet—Create a Persistent Set Collection” on page 1006
- “TO2\_removeValue—Remove Value” on page 1088.

## T02\_maxEntry–Return the Maximum Number of Elements in a Collection

This function returns the maximum number of entries for the log or keyed log.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_maxEntry (const T02_PID_PTR pid_ptr,
                   T02_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to the location where the persistent identifier (PID) assigned to the collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The maximum number of entries for the log or keyed log will be returned as the return value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
T02_ERROR_NOT_INIT
```

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example returns the maximum number of entries.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */
T02_ENV_PTR env_ptr;       /* PTR to the T02 environment */
T02_PID_PTR collect;       /* PTR to PID return field */
:
:
if ((maxentries = T02_maxEntry(pid_ptr,
                              env_ptr)) == T02_ERROR)
{
    printf("ERROR...T02_maxEntry failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_maxEntry successful!\n");
    printf("Returned Entries count is %d\n", maxentries);
}
```



## Related Information

None.

## T02\_readOnly—Get the Read-Only Attribute of the Collection

This function returns the state of the read-only attribute for the specified persistent identifier (PID).

### Format

```
#include <c$to2.h>
BOOL T02_readOnly (const T02_PID_PTR  pid_ptr,
                    T02_ENV_PTR      env_ptr);
```

#### pid\_ptr

The pointer to the PID of the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

#### T02\_IS\_TRUE

The collection is marked read-only.

#### T02\_IS\_FALSE

The collection is not marked read-only.

### Error Return

An error return is indicated by a zero return code. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For the T02\_IS\_FALSE result, check the TPFCS error code by entering T02\_getErrorCode to distinguish this result from an error return indication. If the error code is zero, the Boolean result is false. Otherwise, an error is indicated and you can retrieve the error text by entering T02\_getErrorText. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_PID
```

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example determines the read-only attribute of a collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */
T02_ENV_PTR      env_ptr;        /* Pointer to T02 Environment */
T02_PID          blob;
:
:
if (T02_ReadOnly(&blob,
                env_ptr) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code == 0)
    {
        printf("T02_ReadOnly successful!\n");
        printf("Collection attribute IS NOT Read-Only \n");
    }
}
else
{

```

```
        printf("TO2_Read_Only failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("TO2_ReadOnly successful!\n");
    printf("Collection attribute IS Read-Only \n");
}
:
```

## Related Information

- “TO2\_getCollectionAccessMode — Retrieve the Access Mode of a Collection” on page 1049
- “TO2\_setReadOnly—Set the Read-Only Attribute of the Collection” on page 1105.

## T02\_removeIndex–Remove Element from the Index

This function removes the specified element from the index.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_removeIndex (const T02_PID_PTR  pid_ptr,
                      T02_ENV_PTR  env_ptr,
                      const long      *index);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that will be removed.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### index

The pointer to the relative element number that will be removed. This number is 1-based.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_INDEX
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

### Programming Considerations

A commit scope will be created for the T02\_removeIndex request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_removeIndex request, the scope will be rolled back.

### Examples

The following example removes the third element in the collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* Pointer to the T02 environment */
T02_PID_PTR  collect;      /* Pointer to PID return field */
long index=3;
:
:

if (T02_removeIndex(&collect,
```

```
env_ptr,  
&index) == T02_ERROR)  
{  
    printf("T02_removeIndex failed!\n");  
    process_error(env_ptr);  
}  
else  
{  
    printf("T02_removeIndex successful!\n");  
}
```

## Related Information

- “T02\_add—Add an Element to a Collection” on page 883
- “T02\_at—Return the Specified Element by Index” on page 903
- “T02\_atPut—Update the Specified Element” on page 913
- “T02\_removeValue—Remove Value” on page 1088.

## T02\_removeKey–Remove the Key-Entry Pair

This function searches the specified collection for the specified key and, if found, the key and the associated element are removed from the collection. If the collection contains duplicate keys equal to the specified key, the first one found will be removed.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_removeKey (const T02_PID_PTR  pid_ptr,
                    T02_ENV_PTR  env_ptr,
                    const void    *key);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that will be removed.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the value that will be used as the key to locate the specific entry. The length of the key is assumed to be equal to the key length value specified when the collection was created.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

### Programming Considerations

A commit scope will be created for the T02\_removeKey request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_removeKey request, the scope will be rolled back.

### Examples

The following example removes the key.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */
T02_ENV_PTR  env_ptr;      /* Pointer to the T02 environment */
T02_PID_PTR  pid_ptr;
```

```
char          key[] = "Some key value of appropriate length";
:
:

if (T02_removeKey(pid_ptr,
                  env_ptr,
                  key) == T02_ERROR)
{
    printf("T02_removeKey failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_removeKey successful!\n");
}
```

## Related Information

- “T02\_add—Add an Element to a Collection” on page 883
- “T02\_at—Return the Specified Element by Index” on page 903
- “T02\_atPut—Update the Specified Element” on page 913
- “T02\_removeValue—Remove Value” on page 1088.

## T02\_removeKeyPath—Remove a Key Path from a Collection

This function allows an application program to remove an alternate key path from a collection. TPF collection support (TPFCS) will delete the key path from the collection and release all the resources of that key path back to the TPF system.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_removeKeyPath (const T02_PID_PTR pid_ptr,
                        T02_ENV_PTR env_ptr,
                        const char *name);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### name

The pointer to a string that is the name of the alternate key path that will be removed from the collection.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_LOCATOR_NOT_FOUND
T02_ERROR_METHOD
T02_ERROR_NOT_INIT
T02_ERROR_PID
T02_ERROR_RESERVED_NAME
T02_ERROR_ZERO_PID
```

## Programming Considerations

- The T02\_removeKeyPath function cannot be used to delete the primary key path.
- This function uses TPF transaction services on behalf of the caller.

## Examples

The following example removes the specified key path from the collection.

```
#include <c$to2.h>           /* T02 API function prototypes */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;     /* Pointer to T02 Environment */
T02_PID        keyset;
char           name="fieldA"; /* name of key path to remove */
:
:
if (T02_removeKeyPath(&keyset,
```



```
env_ptr,  
name) == T02_ERROR)  
{  
    printf("T02_removeKeyPath failed!\n");  
    process_error(env_ptr);  
}  
else  
{  
    printf("T02_removeKeyPath was successful!\n");  
}
```

## Related Information

- “TO2\_addKeyPath—Add a Key Path to a Collection” on page 890
- “TO2\_getCurrentKey—Retrieve the Current Key” on page 1141
- “TO2\_setKeyPath—Set a Cursor to Use a Specific Key Path” on page 1180.

## T02\_removeRBA—Remove an Area from a BLOB

This function logically zeros the area of the binary large object (BLOB) starting at the specified relative byte address (RBA) for the specified length. The function causes no change in the size of the BLOB.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_removeRBA (const T02_PID_PTR pid_ptr,
                    T02_ENV_PTR env_ptr,
                    const long *relativeByteToRemove,
                    const long *lengthToRemove);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the area in a BLOB that will be removed.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### relativeByteToRemove

The pointer to the start of the area that will be removed.

#### lengthToRemove

The pointer to the number of bytes that will be removed.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_INVALID_INDEX
T02_ERROR_INVALID_DATA_LGH
```

### Programming Considerations

A commit scope will be created for the T02\_removeRBA request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_removeRBA request, the scope will be rolled back.

### Examples

The following example zeros 20 bytes in the BLOB specified, starting at relative byte address 10.

```
#include <c$to2.h>          /* Needed for T02 API functions      */
#include <stdio.h>          /* APIs for standard I/O functions */

long test_error(void);

T02_ENV_PTR env_ptr;      /* PTR to the T02 environment      */
T02_PID_PTR pid_ptr;      /* PTR to PID return field         */
long rba=10;              /* relative byte to remove         */
```

```
long          len=20;      /* pointer to Number Bytes to Remove */
:
:
if (TO2_removeRBA(pid_ptr,
                  env_ptr,
                  &rba,
                  &len) == T02_ERROR)
{
    printf("TO2_removeRBA failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_removeRBA successful!\n");
}
```

## Related Information

- “TO2\_atRBA–Retrieve Data from a BLOB” on page 915
- “TO2\_atRBAPut–Store Data in a BLOB” on page 917
- “TO2\_atRBAWithBuffer–Retrieve Data from a BLOB” on page 920.

## T02\_removeRecoupIndexFromPID—Remove a PID to Index Association

This function deletes a recoup index to collection association.

### Format

```
#include <c$to2.h>
long T02_removeRecoupIndexFromPID (const T02_PID_PTR pid_ptr,
                                     T02_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to the PID of the collection whose recoup index association will be removed.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

```
T02_ERROR_ENV
T02_ERROR_PID
```

## Programming Considerations

- This function uses TPF transaction services on behalf of the caller.
- If a collection is not associated with a recoup index, a remove will return a positive value.

## Examples

The following example removes the association between a collection and a recoup index.

```
#include <c$to2.h>          /* Needed for T02 API Functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* Pointer to T02 environment */
T02_PID_PTR    collect;    /* target object's PID */
:
:

if (T02_removeRecoupIndexFromPID(&collect,
                                env_ptr) == T02_ERROR)
{
    printf ("T02_removeRecoupIndexFromPID failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_removeRecoupIndexFromPID successful\n");
```

## Related Information

- “T02\_associateRecoupIndexWithPID—Create a PID to Index Association” on page 899
- “T02\_createRecoupIndex—Create a Recoup Index” on page 998

## **TO2\_removeRecoupIndexFromPID**

- "TO2\_deleteRecoupIndex-Delete a Recoup Index" on page 1040.

## T02\_removeValue–Remove Value

This function removes the first element whose length and value exactly match the specified value from the collection.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_removeValue (const T02_PID_PTR  pid_ptr,
                       T02_ENV_PTR      env_ptr,
                       const void        *value,
                       const long        *valueLength);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that will be removed.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### value

The pointer to the value that will be removed from the collection.

#### valueLength

The pointer to the length of the value.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

### Programming Considerations

A commit scope will be created for the T02\_removeValue request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_removeValue request, the scope will be rolled back.

### Examples

The following example removes an element from a collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;       /* Pointer to the T02 environment */
```

```

T02_PID      bag;
char         bagdata[] = "Bag:Collection:Element:.Data"; /* data */
long        bagdatalength; /* length */
:
:
if (T02_removeValue(&bag,
                    env_ptr,
                    &bagdata,
                    &bagdatalength) == T02_ERROR)
{
    printf("T02_removeValue failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_removeValue successful!\n");
}

```

## Related Information

- “TO2\_add—Add an Element to a Collection” on page 883
- “TO2\_includes—Search Collection for Element Equal to the Argument” on page 1064
- “TO2\_removeValueAll—Remove All Matching Values from the Collection” on page 1090.

## T02\_removeValueAll–Remove All Matching Values from the Collection

This function removes all the elements whose length and value exactly match the specified value from the collection.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_removeValueAll (const T02_PID_PTR    pid_ptr,
                        T02_ENV_PTR    env_ptr,
                        const void      *value,
                        const long      *valueLength);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that will be removed.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### value

The pointer to the value that will be removed from the collection.

#### valueLength

The pointer to the length of the value.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

### Programming Considerations

A commit scope will be created for the T02\_removeValueAll request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_removeValueAll request, the scope will be rolled back.

### Examples

The following example removes any remaining value element.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* Pointer to the T02 environment */
```



```

T02_PID      bag;
char         bagdata[] = "Bag:Collection:Element:Data"; /* data    */
long        bagdatalength; /* length */
:
:
if (T02_removeValueAll(&bag,
                      env_ptr,
                      bagdata,
                      &bagdatalength) == T02_ERROR)
{
    printf("T02_removeValueAll failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_removeValueAll successful!\n");
}

```

### Related Information

- “TO2\_add—Add an Element to a Collection” on page 883
- “TO2\_includes—Search Collection for Element Equal to the Argument” on page 1064
- “TO2\_removeValue—Remove Value” on page 1088.

## T02\_replaceBLOB—Replace the Contents of a BLOB with New Data

This function allows an application program to replace the contents of a binary large object (BLOB) with an updated copy of the contents. The size of the BLOB will be made either smaller or larger depending on the content size passed as a parameter.

**Note:** This function only supports BLOB collections.

### Format

```
#include <c$to2.h>
long T02_replaceBLOB (const T02_PID_PTR    pid_ptr,
                      T02_ENV_PTR        env_ptr,
                      const void          *data,
                      const long          *dataLgh,
                      const long          *sequenceCounter);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### data

The pointer to the new data to write to the BLOB. This data will replace the current contents of the BLOB.

#### dataLgh

The pointer to a field that contains the length of the new data.

#### sequenceCounter

The pointer to a field that contains the sequence counter returned when the BLOB was read to retrieve the last valid contents of the BLOB.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LGH
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_NOT_INIT
T02_ERROR_PID
T02_ERROR_SEQCTR
T02_ERROR_UPDATE_NOT_ALLOWED
T02_ERROR_ZERO_PID
```

### Programming Considerations

- A commit scope will be created for the T02\_replaceBLOB function. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_replaceBLOB request, the scope will be rolled back.

- Do not use the TO2\_replaceBLOB function to give a BLOB a zero size by specifying a zero data length. To assign a zero size to a BLOB, use the TO2\_setSize or TO2\_makeEmpty function.

## Examples

The following example replaces the contents of a BLOB with a single byte.

```
#include <c$to2.h>           /* T02 API function prototypes */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;      /* Pointer to T02 Environment */
T02_PID        blob;
long           seqCtr;       /* field to hold sequence counter */
long           dataLgh;      /* field to hold length of BLOB */
char           * bufPtr;

:
:
if (bufPtr = (T02_getBLOB(&blob,
                        env_ptr,
                        &dataLgh,
                        &seqCtr)) == T02_ERROR)
{
    printf("T02_getBLOB failed!\n");
    process_error(env_ptr);
}

*bufPtr = 'A';
dataLgh=1;

if (T02_replaceBLOB(&blob,
                  env_ptr,
                  bufPtr,
                  &dataLgh,
                  &seqCtr) == T02_ERROR)
{
    printf("T02_replaceBLOB failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_replaceBLOB successful!\n");
}
```

## Related Information

- “TO2\_atRBA–Retrieve Data from a BLOB” on page 915
- “TO2\_atRBAPut–Store Data in a BLOB” on page 917
- “TO2\_getBLOB–Retrieve the Contents of a BLOB” on page 1045
- “TO2\_getBLOBwithBuffer–Retrieve Contents of a BLOB Using a Passed Buffer” on page 1047
- “TO2\_removeRBA–Remove an Area from a BLOB” on page 1084
- “TO2\_size–Return the Size of the Collection” on page 1109
- “TO2\_writeNewBLOB–Create a New BLOB and Add the Passed Data” on page 1111.

## T02\_restore—Restore a Previously Captured Collection

This function restores the specified collection to the database using its previously assigned data definition (DD) values and assigns a new persistent identifier (PID) to the collection. If the collection also has assigned properties, the properties will also be restored as part of the T02\_restore request.

### Format

```
#include <c$to2.h>
T02_BUF_PTR T02_restore (const T02_PID_PTR  pid_ptr,
                           T02_ENV_PTR    env_ptr,
                           T02_PID_PTR    newPidPtr,
                           const void      *data,
                           const long      *dataLength,
                           const long      *mountTimeOut);
```

#### pid\_ptr

The pointer to the PID of the captured collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### newPidPtr

The pointer to the location where the PID assigned to the restored collection will be returned.

#### data

The positioning data from the T02\_capture function call.

#### dataLength

The pointer to an area that contains the length of the positioning data.

#### mountTimeOut

The pointer to an area that contains the number of seconds to wait for the mount of the external device.

### Normal Return

For a normal return, the T02\_restore function will return a buffer item that contains the user data written as part of the T02\_capture request. The normal return is a pointer (T02\_BUF\_PTR) to a structure (buffer) of type T02\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, the T02\_getErrorCode function can be used to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_DATA_LGH
T02_ERROR_NO_XTERNAL_DEFINED
T02_ERROR_NO_XTERNAL_DEVICES
T02_ERROR_PERMANENT_XTERNAL
T02_ERROR_TAPE_FORMAT
T02_ERROR_TIMEOUT
T02xd_ERROR_tapeBlocked
```

## Programming Considerations

- Collections cannot be captured or restored using blocked tapes.
- A commit scope will be created for the T02\_restore request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_restore request, the scope will be rolled back.
- This function uses the information returned to the buffer on the previous T02\_capture request.

## Examples

The following example restores a persistent collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */
T02_BUF_PTR    buffer_ptr;       /* Pointer to buffer */
T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
T02_PID        collect;          /* will hold collection's PID */
T02_PID        newpid;           /* will hold collection's PID */
char           *positionData;    /* */
long           positionDataLength; /* */
long           timeOut = 5*60;    /* time out mount in 5 minutes */

:
buffer_ptr=T02_restore (&collect,
                        env_ptr,
                        &newpid,
                        positionData,
                        &positionDataLength;
                        &timeOut);
{
    printf("T02_restore failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_restore successful!\n");
```

## Related Information

- “T02\_capture—Capture a Collection to an External Device” on page 924
- “T02\_restoreAsTemp—Restore a Collection as a Temporary Collection” on page 1096
- “T02\_restoreWithOptions—Restore a Collection Using the Specified Options” on page 1098.

## T02\_restoreAsTemp—Restore a Collection as a Temporary Collection

This function restores the specified collection to the database using default temporary data definition (DD) values and assigns the database a new persistent identifier (PID). A temporary collection will exist only for the life of the entry control block (ECB) and is lost when the ECB exits. If the collection also has assigned properties, the properties will also be restored as a temporary collection as part of the T02\_restoreAsTemp request.

### Format

```
#include <c$to2.h>
T02_BUF_PTR T02_restoreAsTemp (const T02_PID_PTR  pid_ptr,
                                   T02_ENV_PTR    env_ptr,
                                   T02_PID_PTR    newPidPtr,
                                   const void     *data,
                                   const long     *dataLength;
                                   const long     *mountTimeOut);
```

#### pid\_ptr

The pointer to the PID of the captured collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### newPidPtr

The pointer to the location where the PID assigned to the restored collection will be returned.

#### data

The positioning data from the T02\_capture function call.

#### dataLength

The pointer to an area that contains the length of the positioning data.

#### mountTimeOut

The pointer to an area that contains the number of seconds to wait for the mount of the external device.

### Normal Return

For a normal return, T02\_restoreAsTemp will return a buffer item that contains the user data written as part of the T02\_capture request. The normal return is a pointer (T02\_BUF\_PTR) to a structure (buffer) of type T02\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, the T02\_getErrorCode function can be used to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_DATA_LGH
T02_ERROR_NO_XTERNAL_DEFINED
T02_ERROR_NO_XTERNAL_DEVICES
T02_ERROR_PERMANENT_XTERNAL
T02_ERROR_TAPE_FORMAT
T02_ERROR_TIMEOUT
T02xd_ERROR_tapeBlocked
```

## Programming Considerations

- Collections cannot be captured or restored using blocked tapes.
- You must enter an explicit T02\_deletePID function call when you have completed processing the returned temporary collection. Do this even though the temporary collection is automatically deleted when the ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.
- A commit scope will be created for the T02\_restoreAsTemp request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_restoreAsTemp request, the scope will be rolled back.

## Examples

The following example restores a persistent collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */
T02_BUF_PTR    buffer_ptr;       /* Pointer to buffer */
T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
T02_PID        collect;          /* will hold collection's PID */
T02_PID        newpid;           /* will hold collection's PID */
char           *positionData;    /* */
long           positionDataLength; /* */
long           timeOut = 5*60;   /* time out mount in 5 minutes */

:
buffer_ptr=T02_restoreAsTemp (&collect,
                             env_ptr,
                             &newpid,
                             positionData,
                             &positionDataLength,
                             &timeOut);
{
    printf("T02_restoreAsTemp failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_restoreAsTemp successful!\n");
```

## Related Information

- “TO2\_capture—Capture a Collection to an External Device” on page 924
- “TO2\_restore—Restore a Previously Captured Collection” on page 1094
- “TO2\_restoreWithOptions—Restore a Collection Using the Specified Options” on page 1098.

## T02\_restoreWithOptions–Restore a Collection Using the Specified Options

This function restores the specified collection to the database using the specified options and assigns a persistent identifier (PID) to the collection. If the collection also has assigned properties, the properties will also be restored as part of T02\_restoreWithOptions request.

### Format

```
#include <c$to2.h>
T02_BUF_PTR T02_restoreWithOptions (const T02_PID_PTR    pid_ptr,
                                     T02_ENV_PTR        env_ptr,
                                     T02_PID_PTR        newPidPtr,
                                     const T02_OPTION_PTR optionListPtr,
                                     const void         *data,
                                     const long          *dataLength,
                                     const long          *mountTimeOut);
```

#### pid\_ptr

The pointer to the PID of the captured collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### newPidPtr

The pointer to the location where the PID assigned to the restored collection will be returned.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList function call or NULL if no options are required.

#### data

The positioning data from a T02\_capture function call.

#### dataLength

The pointer to an area that contains the length of the positioning data.

#### mountTimeOut

The pointer to an area that contains the number of seconds to wait for the mount of the external device.

### Normal Return

For a normal return, T02\_restoreWithOptions will return a buffer item that contains the user data written as part of the T02\_capture request. The normal return is a pointer (T02\_BUF\_PTR) to a structure (buffer) of type T02\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_DATA_LGH
T02_ERROR_DD_NOT_FOUND
T02_ERROR_NO_XTERNAL_DEFINED
T02_ERROR_NO_XTERNAL_DEVICES
```



```

T02_ERROR_NO_PERMANENT_XTERNAL
T02_ERROR_TAPE_FORMAT
T02_ERROR_TIMEOUT
T02xd_ERROR_tapeBlocked

```

## Programming Considerations

- Collections cannot be captured or restored using blocked tapes.
- A commit scope will be created for the T02\_restoreWithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_restoreWithOptions request, the scope will be rolled back.
- This function uses the information returned to the buffer on the previous T02\_capture request.

## Examples

The following example restores a persistent collection.

```

#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */
T02_BUF_PTR    buffer_ptr;       /* Pointer to the buffer */
T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
T02_PID        collect;          /* will hold collection's PID */
T02_PID        newpid;           /* will hold collection's PID */
u_char         ddname[] = "NAME_OF_USER_DD_DEFINITION ";
char           *positionData;
long           positionDataLength;
T02_OPTION_PTR optionListPtr;
long           timeOut = 5*60;   /* time out mount in 5 minutes */

:
:                               /* invoke T02 to create opt list */
optionListPtr=T02_createOptionList(env_ptr,
                                   T02_OPTION_LIST_CREATE, /* create options */
                                   T02_CREATE_SHADOW,      /* use shadowing */
                                   T02_CREATE_DD,          /* use provided ddname*/
                                   T02_OPTION_LIST_END,     /* end of options */
                                   ddname);                /* address of ddname */

if (optionListPtr == T02_ERROR)
{
    process_error(env_ptr);
    return;
}
buffer_ptr=T02_restoreWithOptions (&collect,
                                   env_ptr,
                                   &newpid,
                                   optionListPtr,
                                   positionData,
                                   &positionDataLength,
                                   &timeOut);

{
    printf("T02_restoreWithOptions failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_restoreWithOptions successful!\n");

```

## Related Information

- “T02\_capture—Capture a Collection to an External Device” on page 924
- “T02\_createOptionList—Create a TPF Collection Support Option List” on page 990

## **TO2\_restoreWithOptions**

- “TO2\_restore—Restore a Previously Captured Collection” on page 1094
- “TO2\_restoreAsTemp—Restore a Collection as a Temporary Collection” on page 1096.

## TO2\_setCollectionAccessMode — Set the Access Mode of a Collection

This function is used to change the access mode of a collection. The access mode is used to determine what class of functions may be applied against a collection.

### Format

```
#include <c$to2.h>
long TO2_setCollectionAccessMode (const      TO2_PID_PTR      pid_ptr,
                                           TO2_ENV_PTR      env_ptr,
                                           enum TO2_COLLECT_MODE * mode);
```

#### pid\_ptr

A pointer to the persistent identifier (PID) of the collection whose access mode will be changed.

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### mode

The pointer to an enumerated data type of TO2\_COLLECT\_MODE which is the new access mode for the collection. The access mode can be used to prevent a collection from being deleted or from being changed. The valid modes are:

##### TO2\_COLLECT\_MODE\_NORMAL

This is the default mode that TPFCS sets all collections to when they are created. The collection can be read, appended to, updated, and deleted.

##### TO2\_COLLECT\_MODE\_NOCHANGE

The collection can be read and deleted. It cannot be appended to or updated.

##### TO2\_COLLECT\_MODE\_NODELETE

The collection can be read, appended to, or updated but, the collection cannot be deleted.

**Note:** The mode applies to the collection and not to its elements so that an application can delete all the elements in a collection with this mode but it cannot delete the actual collection itself.

##### TO2\_COLLECT\_MODE\_READONLY

The collection can only be read. It cannot be updated, deleted, or appended to.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

- TO2\_ERROR\_PID
- TO2\_ERROR\_ZERO\_PID
- TO2\_ERROR\_DELETED\_PID
- TO2\_ERROR\_NOT\_INIT
- TO2\_ERROR\_PARAMETER

## TO2\_setCollectionAccessMode

### Programming Considerations

- For information about how to specifically set the access mode to TO2\_COLLECT\_MODE\_READONLY, see “TO2\_setReadOnly–Set the Read-Only Attribute of the Collection” on page 1105.
- The recommended way to mark a collection as read-only is by using the TO2\_setCollectionAccessMode function.
- Using the TO2\_setCollectionAccessMode function to set the access mode of a collection will override a TO2\_setReadOnly setting.

### Examples

The following example sets the access mode of a collection to prevent it from being deleted.

```
#include <c$to2.h>                /* Needed for TO2 API functions */

long  setNoDeleteForCollect(const T02_PID_PTR  pid_ptr,
                             T02_ENV_PTR      env_ptr);

:
/*
/*  issue To2_setCollectionAccessMode call to set the
/*  collection's access mode to NODELETE.
/*  '1' - successful, '-1' - error on request.
/*
/*

long  setNoDeleteForCollect(const T02_PID_PTR  pid_ptr,
                             T02_ENV_PTR      env_ptr)
{
    T02_COLLECT_MODE  mode=TO2_COLLECT_MODE_NODELETE;

    if (TO2_setCollectionAccessMode(pid_ptr,
                                    env_ptr,
                                    &mode) == TO2_ERROR)
    {
        process_error(env_ptr);
        return -1;
    }
    return 1;
}
```

### Related Information

- “TO2\_getCollectionAccessMode — Retrieve the Access Mode of a Collection” on page 1049
- “TO2\_setReadOnly–Set the Read-Only Attribute of the Collection” on page 1105.

## TO2\_setDRprotect — Set Dirty-Reader Protection On

This function activates dirty-reader protection for the collection. Dirty-reader protection determines whether TPF collection support (TPFCS) will use the FILNC macro or the FILEC macro to file records. If dirty-reader protection is on, TPFCS will use the FILNC macro to file records. TPF transaction services maintains a copy of each record filed using a FILNC macro. Maintaining a copy of each record filed can cause the size of TPF transaction services to be enlarged. It can also cause TPF transaction services to either exceed the maximum allowed size or to actually run out of TPF transaction services buffers. So, dirty-reader protection should **only** be set on a collection that will be accessed by dirty readers. When dirty-reader protection is turned on, it **cannot** be turned off.

### Format

```
#include <c$to2.h>
long TO2_setDRprotect(const TO2_PID_PTR pid_ptr,
                      TO2_ENV_PTR env_ptr);
```

#### pid\_ptr

A pointer to the persistent identifier (PID) of the collection whose dirty-reader protection state is to be set on.

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

- TO2\_ERROR\_PID
- TO2\_ERROR\_ZERO\_PID
- TO2\_ERROR\_DELETED\_PID
- TO2\_ERROR\_NOT\_INIT

### Programming Considerations

To avoid running out of TPF transaction services buffers or exceeding the maximum allowed size, consider the following:

- Do not use TPF transaction services. You will need to decide if this is an option.
- Use TPF transaction services, but keep the number of updates low so that the number of records updated in the commit scope will be small. This may be a better option, but it is hard to predict the number of records that will be updated on any specific call to TPFCS.
- Do not use dirty readers and, therefore, dirty-reader protection is not needed. This is the TPFCS default mode. Whenever a collection is created, it is created with dirty-reader protection turned off. Therefore all files are filed using FILEC instead of FILNC.

## T02\_setDRprotect

- If dirty readers are needed, then enable dirty-reader protection only on those collections that will actually be read without locks. Do this by issuing the T02\_setDRprotect function after the collection has been created.
- Once dirty-reader protection has been turned on, it cannot be turned off.

## Examples

The following example activates dirty-reader protection for the provided collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */
T02_PID                    pid_ptr;
T02_ENV_PTR                env_ptr;
:
:
/*****
/*   activate dirty-reader protection for the specified
/*   collection.
/*   '1' - successful, '-1' - error on request.
*****/
if (T02_setDRprotect(pid_ptr,
                    env_ptr) == T02_ERROR)
{
    printf("T02_setDRprotect failed!\n");
    process_error(env_ptr);
    return -1;
}
else
    printf("T02_setDRprotect successful!\n");
return 1;
```

## Related Information

“T02\_getDRprotect — Retrieve Dirty-Reader Protection Status” on page 1055.

## T02\_setReadOnly—Set the Read-Only Attribute of the Collection

This function marks a persistent identifier (PID) as read-only. Once this method has been activated for a PID, the T02\_restore function will no longer work and the PID can never be replaced, added to, or updated. It can be deleted by removing the PID.

### Format

```
#include <c$to2.h>
long T02_setReadOnly (const T02_PID_PTR  pid_ptr,
                        T02_ENV_PTR     env_ptr);
```

#### pid\_ptr

The pointer to the PID of the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero return code. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_PID
```

## Programming Considerations

- A commit scope will be created for the T02\_setReadOnly request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_setReadOnly request, the scope will be rolled back.
- The recommended way to mark a collection as read-only is by using the T02\_setCollectionAccessMode function.
- Using the T02\_setCollectionAccessMode function to set the access mode of a collection will override a T02\_setReadOnly setting.

## Examples

The following example sets the read-only attribute for a collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */
T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */
T02_PID        blob;
:
:
if (T02_setReadOnly(&blob,
                    env_ptr) == T02_ERROR)
{
    printf("T02_setReadOnly failed!\n");
    process_error(env_ptr);
}
else
```

## TO2\_setReadOnly

```
printf("TO2_setReadOnly successful!\n");  
:
```

## Related Information

- “TO2\_deletePID—Delete a Persistent Identifier and Its Backing Store” on page 1352
- “TO2\_readOnly—Get the Read-Only Attribute of the Collection” on page 1076
- “TO2\_setCollectionAccessMode — Set the Access Mode of a Collection” on page 1101.



## T02\_setSize—Set the Size of the BLOB

This function sets the size of an existing binary large object (BLOB). The size of the BLOB can be made either smaller or larger. If the size is made smaller, the associated backing store will be released and cannot be reclaimed at a later date. If the BLOB is made larger, all of the bytes from the old end of BLOB to the new end of BLOB will be read as zero bytes.

**Note:** This function does not support all collections. See Table 47 on page 881 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_setSize (const T02_PID_PTR  pid_ptr,
                  T02_ENV_PTR      env_ptr,
                  const long        *newSize);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment collection as returned by the T02\_createEnv function.

#### newSize

A pointer to the number of bytes to set as the new size of the BLOB.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

## Programming Considerations

A commit scope will be created for the T02\_setSize function. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_setSize request, the scope will be rolled back.

### Examples

The following example sets the size of a BLOB to be 64 bytes.

```
#include <c$to2.h>          /* T02 API function prototypes */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* Pointer to T02 Environment */
T02_PID      blob;
```

## T02\_setSize

```
long          size = 64;
...
if (T02_setSize(&blob,
               env_ptr,
               &size) == T02_ERROR)
{
    printf("T02_setSize failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_setSize successful!\n");
}
```

## Related Information

- “T02\_atRBA—Retrieve Data from a BLOB” on page 915
- “T02\_atRBAPut—Store Data in a BLOB” on page 917
- “T02\_atRBAWithBuffer—Retrieve Data from a BLOB” on page 920
- “T02\_removeRBA—Remove an Area from a BLOB” on page 1084
- “T02\_size—Return the Size of the Collection” on page 1109.

## TO2\_size—Return the Size of the Collection

This function obtains the size of a collection or the number of entries contained in a collection.

### Format

```
#include <c$to2.h>
long TO2_size (const TO2_PID_PTR pid_ptr,
               TO2_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment block as returned by the TO2\_createEnv function.

### Normal Return

#### size

The size of the collection. The value returned for all collections except a BLOB is the number of entries. For a BLOB, the value returned is the number of bytes.

**Note:** For an array collection, not all entries may actually be in use.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine if the zero return code is the actual size of the collection or if the zero return indicates that an error has occurred. If the TO2\_getErrorCode function also returns a zero, the zero return is the size of the collection. Otherwise, the zero return was an error return. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_PID
```

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example determines the size of a specified collection.

```
#include <c$to2.h>           /* Needed for TO2 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

TO2_ENV_PTR    env_ptr;     /* Pointer to TO2 Environment */
TO2_PID        blob;
long           size;

TO2_ERR_CODE    err_code;    /* TO2 error code value */
TO2_ERR_TEXT_PTR err_text_ptr; /* TO2 error code text pointer */
:
if ((size = TO2_size(&blob,
                    env_ptr)) == TO2_ERROR)
{
    err_code=TO2_getErrorCode(env_ptr);
    if (err_code == TO2_IS_FALSE)
        printf("The collection is empty\n");
}
```

## T02\_size

```
        else
        {
            printf("T02_size failed\n");
            process_error(env_ptr);
        }
    }
    else
    {
        printf("T02_size successful!\n");
        printf("Collection size is %d\n",size);
    }
}
```

## Related Information

“T02\_setSize—Set the Size of the BLOB” on page 1107.

## T02\_writeNewBLOB—Create a New BLOB and Add the Passed Data

This function allows an application program to create and populate a new binary large object (BLOB) collection with the passed data. If the **data** and **dataLength** parameters are specified as NULL pointers, TPF collection support (TPFCS) will create and return an empty BLOB.

**Note:** This function only supports BLOB collections.

### Format

```
#include <c$to2.h>
long T02_writeNewBLOB (      T02_PID_PTR      pid_ptr,
                             T02_ENV_PTR      env_ptr,
                             const T02_OPTION_PTR optionListPtr,
                             const void      *data,
                             const long      *dataLength);
```

#### pid\_ptr

The pointer to a field where the new persistent identifier (PID) of the collection will be returned by TPFCS.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList function call or NULL if no options are required. The option list is used to specify create options to be applied by TPFCS when TPFCS creates the new BLOB collection.

#### data

The pointer to the new data that will be written to the BLOB, or NULL to create an empty BLOB.

#### dataLength

The pointer to a field that contains the length of the new data, or NULL to create an empty BLOB. Both the **data** and **dataLength** parameters must be specified as NULL to create an empty BLOB.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DATA_LENGTH
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_NOT_INIT
```

### Programming Considerations

- A commit scope will be created for the T02\_writeNewBLOB function. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_writeNewBLOB request, the scope will be rolled back.

## TO2\_writeNewBLOB

- The TO2\_writeNewBLOB function can be used to create an empty BLOB by specifying both the **data** and **dataLength** parameters as NULL. Both parameters must be NULL or an error will be returned to the caller.

## Examples

The following example causes a new BLOB to be created and loaded with the passed data.

```
#include <c$to2.h>           /* T02 API function prototypes */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;     /* Pointer to T02 Environment */
T02_PID        blob;
long           dataLength=400; /* length of data to store in BLOB */
u_char         data[400];    /* data to store in new BLOB */

/* The initialization of the data field is not shown */

:
if (TO2_writeNewBLOB(&blob,
                    env_ptr,
                    NULL,
                    data,
                    &dataLength) == T02_ERROR)
{
    printf("TO2_writeNewBLOB failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_writeNewBLOB successful!\n");
}
```

## Related Information

- “TO2\_atRBA–Retrieve Data from a BLOB” on page 915
- “TO2\_atRBAPut–Store Data in a BLOB” on page 917
- “TO2\_getBLOB–Retrieve the Contents of a BLOB” on page 1045
- “TO2\_getBLOBwithBuffer–Retrieve Contents of a BLOB Using a Passed Buffer” on page 1047
- “TO2\_removeRBA–Remove an Area from a BLOB” on page 1084
- “TO2\_replaceBLOB–Replace the Contents of a BLOB with New Data” on page 1092
- “TO2\_size–Return the Size of the Collection” on page 1109.

# TPF Collection Support: Cursors

This chapter provides information about TPF collection support (TPFCS) for cursors.

## Supported Collection Classes for Cursor APIs

Table 48 lists the supported collection classes and cursor types for cursor application programming interfaces (APIs). These APIs can only be used with cursors. An X indicates support for that class or cursor type. The following key lists the collection class names and cursor types.

### Key

Symbol	Collection	Symbol	Collection	Symbol	Collection
ARR	Array	KS	Key Set	SB	Sorted Bag
BAG	Bag	KSB	Key Sorted Bag	SEQ	Sequence
BLB	BLOB	KSS	Key Sorted Set	SET	Set
KB	Key Bag	Log	Log	SS	Sorted Set
KL	Keyed Log				

Symbol	Type of Cursor
C	These APIs are supported with nonlocking cursors.
RW	These APIs are supported with locking cursors.

### Notes:

1. If both the C and RW columns have an X, both types of cursors are supported for that API.
2. To create a read-only cursor, use the TO2\_createCursor function (see “TO2\_createCursor—Create a Nonlocking Cursor” on page 1129).
3. To create a locking cursor, use the TO2\_createReadWriteCursor function (see “TO2\_createReadWriteCursor—Create a Locking Cursor” on page 1131).

Table 48. Collection Support: Cursor APIs

C	RW	API Name	ARR	BAG	BLB	KB	KL	KS	KSB	KSS	LOG	SB	SEQ	SET	SS
	X	TO2_addAtCursor											X		
X	X	TO2_atCursor	X	X	X	X	X	X	X	X	X	X	X	X	X
	X	TO2_atCursorPut	X		X	X		X	X	X		X	X		X
X	X	TO2_atCursorWithBuffer	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_atEnd	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_atLast	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_cursorMinus	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_cursorPlus	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_deleteCursor	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_first	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_getCurrentKey				X		X	X	X		X			X
X	X	TO2_getCurrentKeyWithBuffer				X		X	X	X		X			X
X	X	TO2_index	X		X		X				X		X		
X	X	TO2_isEmpty	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_key				X		X	X	X		X			X
X	X	TO2_keyWithBuffer				X		X	X	X		X			X

Table 48. Collection Support: Cursor APIs (continued)

C	RW	API Name	ARR	BAG	BLB	KB	KL	KS	KSB	KSS	LOG	SB	SEQ	SET	SS
X	X	TO2_last	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_locate		X		X		X	X	X		X		X	X
X	X	TO2_more	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_next	X	X	X	X	X	X	X	X	X	X	X	X	X
	X	TO2_nextPut	X		X		X				X		X		
X	X	TO2_nextRBAfor			X										
X	X	TO2_nextWithBuffer	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_peek	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_peekWithBuffer	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_previous	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_previousWithBuffer	X	X	X	X	X	X	X	X	X	X	X	X	X
	X	TO2_remove	X	X	X	X		X	X	X		X	X	X	X
X	X	TO2_reset	X	X	X	X	X	X	X	X	X	X	X	X	X
X	X	TO2_setKeyPath				X		X	X	X		X			X
X	X	TO2_setPositionIndex	X		X		X				X		X		
X	X	TO2_setPositionValue		X		X		X	X	X		X		X	X



## T02\_addAtCursor—Insert an Element in a Sequence Collection

This function inserts the specified data in the collection at the current position of the cursor. Inserting an element into a sequence collection causes the relative positions of all elements following the inserted element to be increased by one position. The T02\_addAtCursor request sets the update sequence counter of the element to zero.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <to2.h>
long T02_addAtCursor (const T02_PID_PTR  cursorPidPtr,
                      T02_ENV_PTR  env_ptr,
                      const void    *data,
                      const long    *dataLength);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### data

The pointer to the element that will be stored in the collection.

#### dataLength

The pointer to an area that contains the maximum length of the element that will be stored. Elements must be less than or equal to the specified size when the collection was created.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_EODAD
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

### Programming Considerations

The T02\_addAtCursor function can only be used to insert an element at the beginning of the collection, between two already existing elements, or as the last element. For example, if you have 10 elements in a collection, you can use the T02\_addAtCursor function to add the first through the 11th element to the collection,

## T02\_addAtCursor

but you could not add the 25th element because no elements exist between 12 and 24. On return, the cursor will be positioned to point to the inserted element. So, if the next call was a T02\_atCursor function, the returned element would be the inserted element.

## Examples

The following example adds an item to a sequence collection at the current position of the cursor.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
T02_PID        cursorPID;        /* PID stored here on cursor */
/* create. */
char           item[] = "Item A"; /* data */
long           itemsiz;
:
:
/*****
/* Make sure that the bag is created before arriving here... */
*****/

itemsiz = sizeof(item);
if (T02_addAtCursor(&cursorPID,
                  env_ptr,
                  item,
                  &itemsiz) == T02_ERROR)
{
    printf("T02_addAtCursor failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_addAtCursor successful!\n");
}
```

## Related Information

- “T02\_atCursor—Return the Element Pointed to by the Cursor” on page 1119
- “T02\_atCursorPut—Store Element Where Cursor Points” on page 1121
- “T02\_remove—Remove the Element Pointed to by the Cursor” on page 1176.

## TO2\_allElementsDo—Iterate over All Elements

This function causes the function pointed at to be called for every element in the collection starting at the current element pointed to by the cursor. The specified user-function will be called with a two-word parameter list. The first word will contain a pointer to a buffer containing the current element. The second word will contain the pointer to the parameter list (**plist** parameter) specified on the TO2\_allElementsDo interface. The TO2\_allElementsDo function will continue to process each element in the collection until either the end of the collection is found or the called function returns a TO2\_ERROR\_EODAD return code to the TO2\_allElementsDo function.

### Format

```
#include <c$to2.h>
long TO2_allElementsDo (const T02_PID_PTR    cursor_ptr,
                          T02_ENV_PTR      env_ptr,
                          long              (*functionPtr) (T02_BUF_PTR, void *)
                          void              *plist);
```

#### cursor\_ptr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### functionPtr

The pointer to the function that will be called for each element of the collection starting at the current cursor position.

#### plist

The pointer to a parameter list that will be passed to the function on every call.

### Normal Return

The normal return is 1.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. If the collection is empty, a +1 is returned. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- You may not reference a function pointer outside the load module where the function is defined unless you use C++ language or you use C language and compile with the DLL option.
- If the user-defined function returns TO2\_ERROR\_EODAD to the TO2\_allElementsDo function, the cursor remains pointing to the last element

## T02\_allElementsDo

processed by the user-defined function. Otherwise, the cursor points to the last element (or beyond the last element) in the collection.

- This function does not use TPF transaction services on behalf of the caller.

## Examples

```
#include <c$to2.h>
#include <stdio.h>

#define ERROR 0
#define OK    1

struct opt { int option1; char option2; };

int example(T02_PID readCursor)
{
    struct opt options = { 5, 'a' };

    /******
    /* Set the cursor to the beginning of the collection.      */
    /******

    if (T02_first(&readCursor, envPtr) == T02_ERROR)
        return ERROR;

    /******
    /* Process all the elements of the collection.              */
    /******

    if (T02_allElementsDo(&readCursor, envPtr, displayLine, &options)
        == T02_ERROR)
    {
        /*-----*/
        /* It's not an error if the collection is empty. */
        /*-----*/

        if (T02_getErrorcode(envPtr) != T02_ERROR_EMPTY)
            return ERROR;
    }
    return OK;
}

/******
/* Function called from T02_allElementsDo to display the data from */
/* the collection.                                                  */
/******

static long displayLine(T02_BUF_PTR buffer, void * plist)
{
    struct opt option_ptr = (struct opt *) plist;

    if (option_ptr->option2 == 'a')
        printf("option a: %s", buffer->data);

    return 0;
}
```

## Related Information

None.

## TO2\_atCursor–Return the Element Pointed to by the Cursor

This function returns the element currently pointed to by the cursor. If the cursor is pointing one position beyond the last element of the collection, a TO2\_ERROR\_EODAD error code will be returned.

**Note:** For array and binary large object (BLOB) collections, the cursor can move more than one position past the last element. Because the cursor position for these two collections would still be valid, the TO2\_atCursor function returns a TO2\_ERROR\_EODAD error code.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_atCursor (const TO2_PID_PTR cursorPidPtr,
                           TO2_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

### Normal Return

The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by NULL. When NULL is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- The caller must free the returned buffer once the caller has stopped using the buffer. Enter a free function to free the buffer.
- The cursor remains positioned at the retrieved element.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example reads the item to which the cursor is currently pointing.

```
#include <c$to2.h>                /* Needed for TO2 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

struct person_record
{
```

## TO2\_atCursor

```
    char          name[20];
    char          address[60];
};
typedef struct person_record PERSON_RECORD;

TO2_PID          cursor;          /* PID of the cursor          */
TO2_ENV_PTR      env_ptr;         /* Pointer to T02 environment */
TO2_BUF_PTR      buffer_ptr;      /* Pointer to element buffer  */
PERSON_RECORD    *person_ptr;     /* Pointer to element data    */
TO2_ERR_CODE     err_code;        /* T02 error code value       */
:
:
/*****
/* Read the item that the cursor is currently pointing to.
*****/
/*****
if ((buffer_ptr = TO2_atCursor(&cursor,
                             env_ptr)) == NULL)
{
    err_code = TO2_getErrorCode(env_ptr);
    if (err_code != TO2_ERROR_EODAD)
    {
        printf("TO2_atCursor failed!\n");
        process_error(env_ptr);
    }
}
else
    printf("Cursor is at the end of the collection!\n");
    return(0);
}
else
{
    person_ptr = (PERSON_RECORD *) buffer_ptr->data;
    printf("TO2_atCursor successful!\n");
}
:
:
free(buffer_ptr);
```

## Related Information

- “TO2\_atCursorPut–Store Element Where Cursor Points” on page 1121
- “TO2\_atCursorWithBuffer–Return the Element Pointed to by the Cursor” on page 1123
- “TO2\_next–Increment and Return the Next Element” on page 1159.

## TO2\_atCursorPut—Store Element Where Cursor Points

This function replaces the element pointed to by the cursor with the specified data. If the cursor is pointing at the end of the last element and the collection is an array or a binary large object (BLOB), it will add the element. For other collections, it will result in an error.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long TO2_atCursorPut (const T02_PID_PTR cursorPidPtr,
                       T02_ENV_PTR env_ptr,
                       const void *data,
                       const long *dataLength,
                       const long *updateSeqCtr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### data

The pointer to the data that will be stored in the element currently pointed to by the cursor.

#### dataLength

The pointer to the area that contains the length of the element that will be stored.

#### updateSeqCtr

The pointer to the area where the update sequence counter from a T02\_atCursor request has been stored when the element was read from the collection.

### Normal Return

The normal return is a positive value. After the request is completed, the cursor points to the next position in the collection.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_SEQCTR
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- When the T02\_atCursorPut function ends, the cursor still points to the same element.
- For keyed collections, only an existing element can be replaced.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example stores a data element where the cursor is currently pointing.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>

T02_PID      cursor;
T02_ENV_PTR  env_ptr;
T02_BUF_PTR  buffer_ptr;
.
.
.
/*****
/* update the data element where the cursor is currently pointing */
*****/
if ( (buffer_ptr=T02_atCursor(&cursor, env_ptr)) == T02_ERROR)
{
    printf("T02_atCursor failed!\n");
    process_error(env_ptr);
}
else
{
    /* modify buffer_ptr->data */
    if (T02_atCursorPut(&cursor,
                        env_ptr,
                        buffer_ptr->data,
                        &buffer_ptr->dataL,
                        &buffer_ptr->updateSeqNbr) == T02_ERROR)
    {
        printf("T02_atCursorPut failed\n");
        process_error(env_ptr);
    }
    else
        printf("T02_atCursorPut successful!\n");
    free(buffer_ptr);
}
```

## Related Information

- “T02\_atCursor—Return the Element Pointed to by the Cursor” on page 1119
- “T02\_nextPut—Store This Element As the Next Element” on page 1161.



## TO2\_atCursorWithBuffer–Return the Element Pointed to by the Cursor

This function returns the element pointed to by the cursor in the specified buffer.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_atCursorWithBuffer (const TO2_PID_PTR  cursorPidPtr
                                     TO2_ENV_PTR    env_ptr,
                                     const long      *bufferLength,
                                     void            *bufferPtr)
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### bufferLength

The pointer to the length of the specified buffer. The buffer should be large enough to hold the data to be returned **plus** 16 additional bytes for a header. The minimum buffer length is 20 bytes.

#### bufferPtr

The pointer to the buffer where the element will be returned.

### Normal Return

The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859). If the length of the data is greater than the length of the buffer, only the amount of data that will fit is placed in the buffer. The actual length of the data element is returned in the **dataL** field of the buffer. Check the returned length against the length of the supplied buffer to determine if the entire element has been returned.

### Error Return

An error return is indicated by NULL. When NULL is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_BUFFER_SIZE
TO2_ERROR_CURSOR
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- The cursor remains positioned at the retrieved element.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example reads the item that the cursor is currently pointing to and returns it in the supplied buffer.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

struct person_record
{
    char        name[20];
    char        address[50];
};
typedef struct person_record PERSON_RECORD;

T02_PID        cursor;      /* Pointer to cursor */
T02_ENV_PTR    env_ptr;     /* Pointer to T02 environment */
T02_BUF_PTR    buffer_ptr;  /* Pointer to collection element */
long           buffer_length; /* Total length of the buffer area */
PERSON_RECORD *person_ptr;  /* Pointer to element data */
T02_ERR_CODE   err_code;    /* T02 error code value */
:
:
/*****
/* Read the item that the cursor is currently pointing to
/* and return it in the supplied buffer.
*****/
buffer_length = sizeof(T02_BUF_HDR) + sizeof(PERSON_RECORD);
if ((buffer_ptr=malloc(buffer_length)) == NULL)
{
    printf("malloc() failed!\n");
    return(0);
}
if ((buffer_ptr=T02_atCursorWithBuffer(&cursor,
                                     env_ptr,
                                     &buffer_length,
                                     buffer_ptr)) == NULL)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_ERROR_EODAD)
    {
        printf("T02_atCursorWithBuffer failed!\n");
        process_error(env_ptr);
    }
    else
        printf("T02_atCursor is at the end of the data!\n");
    }
    else
    {
        person_ptr = buffer_ptr->data;
        if (buffer_ptr->dataL < (sizeof(T02_BUF_HDR) + sizeof(PERSON_RECORD))
            printf("Data returned did not fit in allocated buffer!\n");
        else
            printf("T02_atCursorWithBuffer successful!\n");
    }
}
```

## Related Information

- “T02\_atCursor–Return the Element Pointed to by the Cursor” on page 1119
- “T02\_atCursorPut–Store Element Where Cursor Points” on page 1121
- “T02\_next–Increment and Return the Next Element” on page 1159
- “T02\_nextWithBuffer–Increment and Return the Next Element” on page 1165.

## TO2\_atEnd—Test If the Cursor Is at the End of the Collection

This function tests to see if the cursor is positioned at the end of the collection and then returns TO2\_IS\_TRUE when the cursor is pointing past the last element in the collection. The TO2\_atEnd function is the reverse of the TO2\_more function.

### Format

```
#include <c$to2.h>
BOOL TO2_atEnd (const TO2_PID_PTR cursorPidPtr,
                 TO2_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

### Normal Return

#### TO2\_IS\_FALSE

The cursor is not pointing beyond the last element in the collection.

#### TO2\_IS\_TRUE

The cursor is pointing beyond the last element in the collection.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. This function must also be used with a TO2\_IS\_FALSE return code to distinguish this return from an error return indication. If the error code returned by TO2\_getErrorCode is TO2\_IS\_FALSE (0), this is the actual return code by the function. Otherwise, an error is indicated. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- This function does not change the position of the cursor.
- The TO2\_atLast request will return TO2\_IS\_TRUE if the cursor is pointing at the last element in the collection.
- The TO2\_atEnd request will return TO2\_IS\_TRUE if the cursor is pointing past the last element in the collection.
- If TO2\_atLast returns TO2\_IS\_TRUE, a TO2\_atCursor will return an element.
- If TO2\_atEnd returns TO2\_IS\_TRUE, a TO2\_atCursor will return TO2\_ERROR\_EODAD.
- This function does not use TPF transaction services on behalf of the caller.
- The TO2\_atEnd request will always return TO2\_IS\_FALSE for log and keyed log collections.

## T02\_atEnd

### Examples

The following example tests to see if the cursor is pointing at the end of the collection.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>

T02_PID          cursor;
T02_ENV_PTR      env_ptr;
T02_ERR_CODE     err_code;      /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
:
:
/*****
/* Is the cursor pointing to the end of the collection? */
*****/
if (T02_atEnd(&cursor,env_ptr)) == T02_IS_FALSE)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_ERROR_FALSE)
    {
        printf("T02_atEnd failed!\n");
        process_error(env_ptr);
    }
}
else
    printf("Cursor is not pointing to the end of the collection.\n");
}
else
    printf("Cursor is pointing to the end of the collection.\n");
```

### Related Information

- “T02\_atLast—Test If the Cursor Points to the Last Element” on page 1127
- “T02\_more—Test for More Elements to Process” on page 1157.

## TO2\_atLast—Test If the Cursor Points to the Last Element

This function determines if the cursor is positioned at the last element of the collection.

### Format

```
#include <c$to2.h>
BOOL TO2_atLast (const TO2_PID_PTR cursorPidPtr,
                 TO2_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

### Normal Return

#### TO2\_IS\_FALSE

The cursor is not pointing at the last element in the collection.

#### TO2\_IS\_TRUE

The cursor is pointing at the last element in the collection.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. This function must also be used with a TO2\_IS\_FALSE return code to distinguish this return from an error return indication. If the error code returned by TO2\_getErrorCode is TO2\_IS\_FALSE (0), this is the actual return code by the function. Otherwise, an error is indicated. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- This function does not change the position of the cursor.
- This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example tests to see if the cursor is pointing at the last element in the collection.

```
#include <c$to2.h>                /* Needed for TO2 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

TO2_PID          cursor;
TO2_ENV_PTR      env_ptr;
TO2_ERR_CODE     err_code;      /* TO2 error code value */
TO2_ERR_TEXT_PTR err_text_ptr; /* TO2 error code text pointer */
:
:
/*****
```

## T02\_atLast

```
/* Is the cursor pointing to the last item in the collection? */
/*****
if (T02_atLast(&cursor,env_ptr) == T02_IS_FALSE)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_ERROR_FALSE)
    {
        printf("T02_atLast failed!\n");
        process_error(env_ptr);
    }
}
else
    printf("Cursor is not pointing to the last item in collection.\n");
}
else
    printf("Cursor is pointing to the last item in the collection.\n");
```

## Related Information

“T02\_atEnd—Test If the Cursor Is at the End of the Collection” on page 1125.

## T02\_createCursor—Create a Nonlocking Cursor

This function creates a nonlocking cursor for the specified collection. This type of cursor is sometimes referred to as a dirty-read cursor because no locking is involved. All applications are responsible for managing concurrent updates.

### Format

```
#include <c$to2.h>
long T02_createCursor (const T02_PID_PTR pid_ptr,
                        T02_ENV_PTR env_ptr,
                        T02_PID_PTR cursorPidPtr);
```

#### **pid\_ptr**

The pointer to the persistent identifier (PID) of the collection that will be accessed.

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **cursorPidPtr**

The pointer to where the PID of the cursor collection that is being created will be returned. The cursor is a temporary collection that is allocated only for the life of the entry control block (ECB) that creates it.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_REREAD
T02_ERROR_ZERO_PID
```

## Programming Considerations

- You must enter a T02\_deleteCursor function call when you have completed using the returned cursor. This allows TPF collection support (TPFCS) to release the resources that are used to hold the target collection available for you.
- A positioning request such as T02\_first is required to position the cursor before it can be used to access an element in the collection.
- You can have multiple nonlocking cursors and a single locking cursor open on the same collection regardless of which type of cursor is created first.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example creates a nonlocking cursor.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                 /* APIs for standard I/O functions */

T02_PID      collect;            /* PID of the collection object */
```

## T02\_createCursor

```
T02_ENV_PTR   env_ptr;           /* Pointer to T02 environment      */
T02_PID       cursor;           /* PID of the cursor object      */
T02_ERR_CODE  err_code;         /* T02 error code value          */
:
:
/*****
/* Place a dirty read cursor on the temporary collection.      */
*****/
if (T02_createCursor(&collect,
                    env_ptr, &cursor) == T02_ERROR)
{
    printf("T02_createCursor failed!\n");
    process_error(env_ptr);
    return(0);
}
else
    printf("T02_createCursor successful!\n");
    err_code = T02_first(&cursor, env_ptr);
    :
    :
    err_code = T02_deleteCursor(&cursor, env_ptr);
```

## Related Information

- “T02\_createReadWriteCursor—Create a Locking Cursor” on page 1131
- “T02\_deleteCursor—Delete a Previously Created Cursor” on page 1137
- “T02\_first—Point Cursor to First Element” on page 1139.



## T02\_createReadWriteCursor—Create a Locking Cursor

This function creates a locking cursor for the specified collection. An exclusive lock will be imposed on the specific collection for the life of the cursor. Only this particular cursor and a nonlocking cursor can update elements in the collection at the same time.

### Format

```
#include <c$to2.h>
long T02_createReadWriteCursor (const T02_PID_PTR pid_ptr,
                                  T02_ENV_PTR env_ptr,
                                  T02_PID_PTR cursorPidPtr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) of the collection that will be accessed.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### cursorPidPtr

The pointer to the cursor PID that is created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs). The cursor collection is a temporary collection and is only valid for the life of the entry control block (ECB) that creates it.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ACCESS_MISMATCH
TO2_ERROR_DELETED_PID
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- You must enter a T02\_deleteCursor function call when you have completed using the returned cursor. This allows TPF collection support (TPFCS) to release the resources that are used to hold the target collection available for you. It will also make the target collection available for other users.

**Note:** If you exit without entering T02\_deleteCursor, the system will take a dump at exit time because you will be exiting while holding an interlock on a resource.

- A positioning request such as T02\_first is required to position the cursor before it can be used to access an element in the collection.
- You can have multiple nonlocking cursors and a single locking cursor open on the same collection regardless of which type of cursor is created first.

## TO2\_createReadWriteCursor

- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example creates a locking cursor.

```
#include <c$to2.h>          /* Needed for T02 API Functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_PID      collect;      /* PID of the collection object */
T02_ENV_PTR  env_ptr;      /* Pointer to T02 environment */
T02_PID      cursor;       /* PID of the cursor object */
T02_ERR_CODE err_code;     /* T02 error code value */
:
:
/*****
/* Place a locking cursor on the temporary collection. */
*****/
if (T02_createReadWriteCursor(&collect,
                             env_ptr, &cursor) == T02_ERROR)
{
    printf("T02_createReadWriteCursor failed!\n");
    process_error(env_ptr);
    return(0);
}
else
{
    printf("T02_createReadWriteCursor successful!\n");
    err_code = T02_first(&cursor, env_ptr);
    :
    :
    err_code = T02_deleteCursor(&cursor, env_ptr);
}
```

## Related Information

- “TO2\_createCursor—Create a Nonlocking Cursor” on page 1129
- “TO2\_deleteCursor—Delete a Previously Created Cursor” on page 1137
- “TO2\_first—Point Cursor to First Element” on page 1139.

## T02\_cursorMinus–Decrement Cursor to Previous Element

This function decrements the cursor to point to the previous element in the collection.

### Format

```
#include <c$to2.h>
long T02_cursorMinus (const T02_PID_PTR cursorPidPtr,
                      T02_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_CURSOR
T02_ERROR_ENV
T02_ERROR_EODAD
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

- This function causes the cursor to be pointed at the element before the current element in the collection. The previous element is determined based on what type of collection is involved.
- If the cursor is currently pointing at the first element in the collection, a T02\_ERROR\_EODAD error code will be returned.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example decrements the cursor to point to the previous element in the collection.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_PID      cursor;
T02_ENV_PTR  env_ptr;
T02_ERR_CODE err_code;          /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
:
:
/*****
/* Decrement cursor to point to previous element in collection. */
*****/
```

## TO2\_cursorMinus

```
if (TO2_cursorMinus(&cursor,
                    env_ptr) == TO2_ERROR)
{
    err_code = TO2_getErrorCode(env_ptr);
    if (err_code != TO2_ERROR_EODAD)
    {
        printf("TO2_cursorMinus failed!\n");
        process_error(env_ptr);
    }
    else
        printf("Cursor is already positioned at the first element.\n");
}
else
    printf("TO2_atCursorMinus successful!\n");
```

## Related Information

“TO2\_cursorPlus—Increment Cursor to Next Element” on page 1135.

## T02\_cursorPlus–Increment Cursor to Next Element

This function increments the cursor to the next element in the collection.

### Format

```
#include <c$to2.h>
long T02_cursorPlus (const T02_PID_PTR cursorPidPtr,
                     T02_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_CURSOR
T02_ERROR_ENV
T02_ERROR_EODAD
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

- This function causes the cursor to be advanced to point to the next element in the collection. The next element is determined based on what type of collection is involved. If the cursor is currently pointing at the last element in the collection, a T02\_ERROR\_EODAD error code will be returned.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example increments the cursor to point to the next element in the collection.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_PID      cursor;
T02_ENV_PTR  env_ptr;
T02_ERR_CODE err_code;      /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
:
:
/*****
/* Increment cursor to point to next element in the collection. */
*****/
if (T02_cursorPlus(&cursor,
                  env_ptr) == T02_ERROR)
```

## T02\_cursorPlus

```
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_ERROR_EODAD)
    {
        printf("T02_cursorPlus failed!\n");
        process_error(env_ptr);
    }
    else
        printf("Cursor is already positioned at the last element.\n");
}
else
    printf("T02_atCursorPlus successful!\n");
```

## Related Information

"T02\_cursorMinus—Decrement Cursor to Previous Element" on page 1133.

## T02\_deleteCursor—Delete a Previously Created Cursor

This function deletes the previously created cursor and releases any locks held on the collection.

### Format

```
#include <c$to2.h>
long T02_deleteCursor (const T02_PID_PTR cursorPidPtr,
                       T02_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example deletes a cursor of any type.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_PID      cursor;        /* PID of the cursor object */
T02_ENV_PTR  env_ptr;       /* Pointer to T02 environment */
:
:
/*****
/* Delete the collection's cursor. */
*****/
if (T02_deleteCursor(&cursor,
                    env_ptr) == T02_ERROR)
{
    printf("T02_deleteCursor failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_deleteCursor successful!\n");
```

## **TO2\_deleteCursor**

### **Related Information**

- “TO2\_createCursor—Create a Nonlocking Cursor” on page 1129
- “TO2\_createReadWriteCursor—Create a Locking Cursor” on page 1131.



## TO2\_first–Point Cursor to First Element

This function points the cursor to the first element in the collection. For log collections, the first element is defined as the oldest element.

### Format

```
#include <c$to2.h>
long TO2_first (const TO2_PID_PTR cursorPidPtr,
                TO2_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- This function causes the cursor to be pointed at the first element in the collection. The first element is determined based on what type of collection is involved.
- For dictionary support, you can set the cursor to reference where the first element will be located even if it is empty. It is your responsibility to place something in the empty location.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example positions the cursor at the first element in the collection.

```
#include <c$to2.h>           /* Needed for TO2 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

TO2_PID cursor;
TO2_ENV_PTR env_ptr;
:
/*****
/* Point the cursor to the first element in the collection. */
*****/
if (TO2_first(&cursor,
             env_ptr) == TO2_ERROR)
{
    printf("TO2_first failed!\n");
}
```

## TO2\_first

```
        process_error(env_ptr);  
    }  
    else  
        printf("TO2_first successful!\n");
```

## Related Information

- “TO2\_last—Point Cursor at Last Element” on page 1153
- “TO2\_reset—Reset Cursor to Point to First Element” on page 1178.

## TO2\_getCurrentKey–Retrieve the Current Key

This function allows an application program to retrieve the value of the key being used to access the data element to which the cursor is pointing. This differs from the T02\_key function, which will always return the primary key. The T02\_getCurrentKey function returns either the primary key (if the primary key path is in use) or the appropriate key if an alternate key path is being used to access the data element.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
T02_BUF_PTR T02_getCurrentKey (const T02_PID_PTR   cursorPid_ptr,
                                T02_ENV_PTR      env_ptr);
```

#### cursorPid\_ptr

The pointer to the persistent identifier (PID) assigned to the target cursor.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a pointer (T02\_BUF\_PTR) to a structure (buffer) of type T02\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- This function does not change the position of the cursor.
- If a key path is not being used, both the T02\_getCurrentKey and T02\_key functions will return the same values.
- It is the responsibility of the caller to free the returned buffer once the caller has stopped using the buffer.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example retrieves the current access key value of the current data element.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                 /* APIs for standard I/O functions */

T02_PID      cursor;
T02_ENV_PTR  env_ptr;
```

## TO2\_first

```
TO2_BUF_PTR    buffer_ptr;    /* Pointer to returned buffer    */
u_char         *key;          /* The key of the element        */
:
:
/*****
/* Retrieve the key from the item pointed to by the cursor.
*****/
if ((buffer_ptr = TO2_getCurrentKey(&cursor,
                                   env_ptr)) == NULL)
{
    printf("TO2_getCurrentkey failed!\n");
    process_error(env_ptr);
}
else
{
    key = buffer_ptr->data;
    printf("TO2_getCurrentKey successful!\n");
}
:
free (buffer_ptr);
```

## Related Information

- “TO2\_addKeyPath—Add a Key Path to a Collection” on page 890
- “TO2\_getCurrentKeyWithBuffer—Retrieve the Current Key in the Buffer” on page 1143
- “TO2\_key—Return Current Key Value” on page 1149
- “TO2\_removeKeyPath—Remove a Key Path from a Collection” on page 1082
- “TO2\_setKeyPath—Set a Cursor to Use a Specific Key Path” on page 1180.

## TO2\_getCurrentKeyWithBuffer–Retrieve the Current Key in the Buffer

This function searches the specified collection for the specified current key and, if found, returns the associated element value in the specified buffer. This differs from the `T02_keyWithBuffer` function, which will always return the primary key. The `T02_getCurrentKeyWithBuffer` function returns either the primary key (if the primary key path is in use) or the appropriate key if an alternate key path is being used to access the data element.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
T02_BUF_PTR T02_getCurrentKeyWithBuffer (const T02_PID_PTR   cursorPid_ptr,
                                          T02_ENV_PTR     env_ptr,
                                          const long        *bufferLength,
                                          T02_BUF_PTR       buffer);
```

#### **cursorPid\_ptr**

The pointer to the persistent identifier (PID) assigned to the target cursor.

#### **env\_ptr**

The pointer to the environment as returned by the `T02_createEnv` function.

#### **bufferLength**

The pointer to the length of the specified buffer.

#### **buffer**

The pointer to the buffer where the key will be returned.

### Normal Return

The normal return is a pointer (`T02_BUF_PTR`) to a structure (buffer) of type `T02_BUF_HDR` (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the `T02_getErrorCode` function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- This function does not change the position of the cursor.
- If a key path is not being used, both the `T02_getCurrentKeyWithBuffer` and `T02_keyWithBuffer` functions will return the same values.
- This function does not use TPF transaction services on behalf of the caller.
- If the length of the key is greater than the length of the buffer, only the amount of data that will fit is placed in the buffer.

## Examples

The following example retrieves the current active key value of the current data element.

```
#include <c$to2.h>           /* Needed for T02 API Functions    */
#include <stdio.h>           /* APIs for standard I/O functions */
#define KEY_SIZE    16

T02_PID      cursor;
T02_ENV_PTR  env_ptr;
T02_BUF_PTR  buffer_ptr; /* Pointer to T02 environment    */
u_char       *key=NULL;   /* The key of the element      */
long         buffer_length;
:
:
:
/*****
/* Retrieve the key from the item pointed to by the cursor
/* and return it in the supplied buffer.
*****/
buffer_length = KEY_SIZE + sizeof(T02_BUF_HDR);
if ((buffer_ptr=malloc(buffer_length)) == NULL)
{
    printf("malloc() failed!\n");
    return(0);
}
if ((buffer_ptr=T02_getCurrentKeyWithBuffer(&cursor,
                                           env_ptr,
                                           &buffer_length,
                                           buffer_ptr)) == NULL)
{
    printf("T02_getCurrentkeyWithBuffer failed!\n");
    process_error(env_ptr);
}
else
{
    key = buffer_ptr->data;
    printf("T02_getCurrentKeyWithBuffer successful!\n");
}
:
:
free (buffer_ptr);
```

## Related Information

- “T02\_addKeyPath—Add a Key Path to a Collection” on page 890
- “T02\_getCurrentKey—Retrieve the Current Key” on page 1141
- “T02\_keyWithBuffer—Return Current Key Value in the Buffer” on page 1151
- “T02\_removeKeyPath—Remove a Key Path from a Collection” on page 1082
- “T02\_setKeyPath—Set a Cursor to Use a Specific Key Path” on page 1180.

## TO2\_index–Return Current Position Index Value

This function returns the current position index value from the cursor. An integer index is returned for arrays, binary large objects (BLOBs), logs, and sequence collections.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long TO2_index (const T02_PID_PTR cursorPidPtr,
                 T02_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is the current position index value (a positive value).

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- This function does not change the position of the cursor.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example retrieves the current position index value from the cursor.

```
#include <c$to2.h>           /* Needed for T02 API Functions      */
#include <stdio.h>           /* APIs for standard I/O functions */
T02_PID cursor;
T02_ENV_PTR env_ptr;
long index_val;
:
:
/*****
/* Retrieve the current position index value from the cursor.      */
*****/
if ((index_val=T02_index(&cursor,
                        env_ptr)) == T02_ERROR)
{
    printf("T02_index failed!\n");
```

## T02\_index

```
        process_error(env_ptr);  
    }  
    else  
        printf("T02_index successful!\n");
```

## Related Information

None.



## T02\_isEmpty–Test If Collection Is Empty

This function determines if the collection contains any elements.

### Format

```
#include <c$to2.h>
BOOL T02_isEmpty (const T02_PID_PTR cursorPidPtr,
                  T02_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

#### TO2\_IS\_FALSE

The collection is not empty.

#### TO2\_IS\_TRUE

The collection is empty.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. This function must also be used with a TO2\_IS\_FALSE return code to distinguish this return from an error return indication. If the error code returned by T02\_getErrorCode is TO2\_IS\_FALSE (0), this is the actual return code by the function. Otherwise, an error is indicated. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- This function does not change the position of the cursor.
- This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example tests the cursor to see if the collection contains any elements.

```
#include <c$to2.h>                                /* Needed for T02 API Functions */
#include <stdio.h>                                  /* APIs for standard I/O functions */

T02_PID      cursor;
T02_ENV_PTR  env_ptr;
T02_ERR_CODE err_code;                             /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr;                      /* T02 error code text pointer */
:
:
/*****
/* Is the cursor pointing to an empty collection?
*****/
```

## T02\_isEmpty

```
if (T02_isEmpty(&cursor,
               env_ptr)) == T02_IS_FALSE)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != 0)
    {
        printf("T02_isEmpty failed!\n");
        process_error(env_ptr);
    }
}
else
    printf("The cursor is not pointing to an empty collection!\n");
}
else
    printf("The cursor is pointing to an empty collection.\n");
```

## Related Information

“T02\_size—Return the Size of the Collection” on page 1109.

## TO2\_key–Return Current Key Value

This function returns the key value for the element pointed to by the cursor. The current key is returned for dictionaries, key bags, key sets, and key sorted bags. The sort field is returned for sorted bag and sorted set collections.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_key (const TO2_PID_PTR cursorPidPtr,
                     TO2_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

### Normal Return

The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- This function does not change the position of the cursor.
- The caller must free the returned buffer once the caller has stopped using the buffer. Enter a free function to free the buffer.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example shows how the key value is retrieved from the item to which the cursor is currently pointing.

```
#include <c$to2.h>           /* Needed for TO2 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

TO2_PID      cursor;
TO2_ENV_PTR  env_ptr;
TO2_BUF_PTR  buffer_ptr;   /* Pointer to returned buffer */
u_char      *key;          /* The key of the element */
TO2_ERR_CODE err_code;     /* TO2 error code value */
TO2_ERR_TEXT_PTR err_text_ptr; /* TO2 error code text pointer */
```

## TO2\_key

```

:
/*****
/* Retrieve the key from the item pointed to by the cursor.      */
*****/
if ((buffer_ptr = TO2_key(&cursor,
                          env_ptr)) == NULL)
{
    printf("TO2_key failed!\n");
    process_error(env_ptr);
}
else
{
    key = buffer_ptr->data;
    printf("TO2_key successful!\n");
}

```

## Related Information

- “TO2\_keyWithBuffer–Return Current Key Value in the Buffer” on page 1151
- “TO2\_size–Return the Size of the Collection” on page 1109.

## TO2\_keyWithBuffer–Return Current Key Value in the Buffer

This function retrieves the key value from the item that the cursor is currently pointing to and returns it in the specified buffer. The current key is returned for dictionaries, key bags, key sets, and key sorted bags.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_keyWithBuffer (const TO2_PID_PTR  cursorPidPtr,
                                   TO2_ENV_PTR    env_ptr,
                                   const long      *bufferLength,
                                   TO2_BUF_PTR    buffer);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### bufferLength

The pointer to the length of the specified buffer. The buffer should be large enough to hold the data to be returned plus 16 additional bytes for a header. The minimum buffer length is 20 bytes.

#### buffer

The pointer to the buffer where the element will be returned.

### Normal Return

For a normal return, this buffer contains a C header file followed by a copy of the data requested. The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

TO2\_ERROR\_BUFFER\_SIZE

### Programming Considerations

- This function does not change the position of the cursor.
- This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example shows how the key value is retrieved from the item to which the cursor is currently pointing and returns it in the specified buffer.

```
#include <c$to2.h>          /* Needed for TO2 API Functions */
#include <stdio.h>          /* APIs for standard I/O functions */
#define KEY_SIZE 16
```

## T02\_keyWithBuffer

```
T02_PID          cursor;
T02_ENV_PTR      env_ptr;
T02_BUF_PTR      buffer_ptr; /* Pointer to T02 environment */
u_char          *key=NULL;   /* The key of the element */
T02_ERR_CODE     err_code;   /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
long             buffer_length;
:
:
/*****
/* Retrieve the key from the item pointed to by the cursor */
/* and return it in the supplied buffer. */
*****/
buffer_length = KEY_SIZE + sizeof(T02_BUF_HDR);
if ((buffer_ptr=malloc(buffer_length)) == NULL)
{
    printf("malloc() failed!\n");
    return(0);
}
if ((buffer_ptr=T02_keyWithBuffer(&cursor,
                                env_ptr,
                                &buffer_length,
                                buffer_ptr)) == NULL)
{
    printf("T02_keyWithBuffer failed!\n");
    process_error(env_ptr);
}
else
{
    if (buffer_length < (KEY_SIZE + sizeof(T02_BUF_HDR)))
    {
        printf("Key returned did not fit in allocated buffer.\n");
    }
    else
    {
        printf("T02_keyWithBuffer successful!\n");
        key = buffer_ptr->data;
    }
}
```

## Related Information

“T02\_key–Return Current Key Value” on page 1149.

## T02\_last–Point Cursor at Last Element

This function points the cursor to the last element in the collection.

### Format

```
#include <c$to2.h>
long T02_last (const T02_PID_PTR cursorPidPtr,
                T02_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

- This function causes the cursor to be pointed at the last element in the collection. The last element is determined based on what type of collection is involved.
- For dictionary support, you can set the cursor to reference where the last element will be located even if it is empty. It is your responsibility to place something in the empty location.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example shows how the cursor is pointed to the last element in the collection.

```
#include <c$to2.h>          /* Needed for T02 API Functions    */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_PID      cursor;
T02_ENV_PTR  env_ptr;
:
:
/*****
/* Point the cursor to the last element in the collection.
*****/
if (T02_last(&cursor,
            env_ptr)) == T02_ERROR)
{
    printf("T02_last failed!\n");
```

## **T02\_last**

```
        process_error(env_ptr);  
    }  
    else  
        printf("T02_last successful!\n");
```

## **Related Information**

"T02\_first—Point Cursor to First Element" on page 1139.



## T02\_locate—Locate Key and Point Cursor to Its Element

This function locates and points the cursor at the element with a key or sort field that matches the argument.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_locate (const T02_PID_PTR  cursorPidPtr,
                  T02_ENV_PTR      env_ptr,
                  const long        *argumentLength,
                  const void        *argument);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### argumentLength

A pointer to a field that contains the length of the argument used to search the collection.

#### argument

A pointer to the argument to search the collection for a match.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- This function uses the current key path field to match the specified value (or the original key or sort field if no key path has been set).
- This function causes the cursor to be pointed at the first element with a key path, key, or sort field that matches the given value. If there is no match, the position of the cursor depends on the collection implementation.
- This function does not use TPF transaction services on behalf of the caller.

## T02\_locate

### Examples

The following example shows how the element is located in the cursor that matches the search key and positions the cursor to point to it.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ERR_CODE    err_code;    /* T02 error code value */
T02_PID         cursor;
T02_ENV_PTR     env_ptr;     /* Pointer to T02 environment */
u_char         argument;
long            arg_Length;
:
:
/*****
/* Locate and point the cursor to the element matching the key. */
*****/
if (T02_locate(&cursor,
              env_ptr,
              &arg_Length,
              &argument) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code == T02_ERROR_LOCATOR_NOT_FOUND)
    {
        printf("There are no elements that match the search criteria.\n");
    }
}
else
{
    printf("T02_locate failed!\n");
    process_error(env_ptr);
}
}
else
{
    printf("T02_locate successful!\n");
}
}
```

### Related Information

- “T02\_setPositionIndex–Point Cursor at Specified Position” on page 1182
- “T02\_setPositionValue–Point Cursor at Specified Element” on page 1184.

## TO2\_more–Test for More Elements to Process

This function tests the cursor position to determine if there are more elements in the collection and then returns TO2\_IS\_FALSE when the cursor is pointing past the last element in the collection. The TO2\_more function is the reverse of the TO2\_atEnd function.

### Format

```
#include <c$to2.h>
BOOL TO2_more (const TO2_PID_PTR cursorPidPtr,
               TO2_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

### Normal Return

#### TO2\_IS\_FALSE

There are no more elements to process (the cursor is not pointing to an element before the last element).

#### TO2\_IS\_TRUE

There are more elements to process (the cursor is pointing to an element before the last element).

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. This function must also be used with a TO2\_IS\_FALSE return code to distinguish this return from an error return indication. If the error code returned by TO2\_getErrorCode is TO2\_IS\_FALSE (0), this is the actual return code by the function. Otherwise, an error is indicated. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- This function does not change the position of the cursor.
- This function does not use TPF transaction services on behalf of the caller.
- The TO2\_more request will always return TO2\_IS\_TRUE for log and keyed log collections.

### Examples

The following tests to see if there are subsequent elements to process after the element that the cursor is currently pointing to.

## T02\_more

```
#include <c$to2.h>          /* Needed for T02 API Functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_PID      cursor;
T02_ENV_PTR  env_ptr;
T02_ERR_CODE err_code;     /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
:
/*****
/* Are there more elements after the current one?
*****/
if (T02_more(&cursor,
            env_ptr)) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_IS_FALSE)
    {
        printf("T02_more failed!\n");
        process_error(env_ptr);
    }
}
else
    printf("There are no more elements after the current element.\n");
}
else
    printf("There are more elements after the current element.\n");
```

## Related Information

"T02\_atEnd-Test If the Cursor Is at the End of the Collection" on page 1125.

## TO2\_next–Increment and Return the Next Element

This function returns the next element in the collection and points the cursor at that element. The definition of the next element depends on the collection implementation.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_next (const TO2_PID_PTR cursorPidPtr,
                      TO2_ENV_PTR env_ptr);
```

#### **cursorPidPtr**

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### **env\_ptr**

The pointer to the environment as returned by the TO2\_createEnv function.

### Normal Return

For a normal return, this buffer contains a C header file followed by a copy of the data requested. The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- The caller must free the returned buffer once the caller has stopped using the buffer. Enter a free function to free the buffer.
- This function leaves the cursor pointing at the returned element. If the cursor is already pointing to the last element in the collection, the TO2\_ERROR\_EODAD return code will be returned. The definition of the next element depends on the collection implementation.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example returns the next element in the collection that the cursor points to.

```
#include <c$to2.h>                /* Needed for TO2 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

struct person_record
{
```

## TO2\_next

```
    char          *name;
    char          *address;
};
typedef struct person_record PERSON_RECORD;

TO2_PID          cursor;      /* PID of the cursor          */
TO2_ENV_PTR      env_ptr;     /* Pointer to T02 environment */
TO2_BUF_PTR      buffer_ptr;  /* Pointer to element buffer   */
PERSON_RECORD    *person_ptr; /* Pointer to element data     */
TO2_ERR_CODE     err_code;    /* T02 error code value       */
TO2_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
:
:
/*****
/* Access and return the next item the cursor points to.
*****/
if ((buffer_ptr = TO2_next(&cursor,
                          env_ptr)) == NULL)
{
    err_code = TO2_getErrorCode(env_ptr);
    if (err_code != TO2_ERROR_EODAD)
    {
        printf("TO2_next failed!\n");
        process_error(env_ptr);
    }
}
else
    printf("The cursor is at the end of the collection!\n");
    return(0);
}
else
{
    person_ptr = (PERSON_RECORD *) buffer_ptr->data;
    printf("TO2_next successful!\n");
}
:
:
free(buffer_ptr);
```

## Related Information

- “TO2\_nextWithBuffer-Increment and Return the Next Element” on page 1165
- “TO2\_peek-Return the Next Element with No Cursor Movement” on page 1168
- “TO2\_previous-Return the Previous Element” on page 1172.

## TO2\_nextPut—Store This Element As the Next Element

This function stores the data as the next element in the collection. For sequence collections, the next element is inserted. For an array or binary large object (BLOB) collection, this function can be used to replace or add the next element.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long TO2_nextPut (const T02_PID_PTR  cursorPidPtr,
                  T02_ENV_PTR      env_ptr,
                  const long        *dataLength,
                  const void        *data);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### dataLength

The length of the element that will be stored at the position after the element pointed to by the cursor.

#### data

The pointer to the actual data element that will be stored.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- When completed successfully for BLOB collections, the cursor will be positioned at the last byte that was added. For all other collections for which this function is supported, the cursor will be positioned at the start of the stored element.
- If T02\_nextPut is called on a cursor positioned with T02\_first on an empty BLOB, T02\_nextPut does not store the data at the first byte, but begins storing it at the second byte.
- This function does not use TPF transaction services on behalf of the caller.

## TO2\_nextPut

### Examples

The following example stores a data element at the next element following where the cursor is currently pointing and then points the cursor to that element.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

struct person_record
{
    char      *name;
    char      *address;
};
typedef struct person_record PERSON_RECORD;

T02_PID      cursor;
T02_ENV_PTR  env_ptr;      /* Pointer to T02 environment */
PERSON_RECORD *person_ptr; /* Pointer to the element data */
long         dataLength;
:
:
/*****
/* Store the data element where the cursor is currently pointing. */
*****/
dataLength = sizeof(PERSON_RECORD);
if (T02_nextPut(&cursor,
                env_ptr,
                (u_char *) person_ptr,
                &dataLength) == T02_ERROR)
{
    printf("T02_nextPut failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_nextPut successful!\n");
```

### Related Information

- “TO2\_atCursorPut–Store Element Where Cursor Points” on page 1121
- “TO2\_next–Increment and Return the Next Element” on page 1159.



## TO2\_nextRBAfor—Return the Next Specified Number of Bytes

This function returns the next specified number of bytes in the collection starting at the current cursor position and updates the cursor to point to the first byte following the returned bytes. The number of bytes read will be the length specified (up to a maximum buffer length of 4 MB (4 194 304)) or the size of the binary large object (BLOB), whichever is smaller.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
void * TO2_nextRBAfor (const T02_PID_PTR  cursorPidPtr,
                        T02_ENV_PTR    env_ptr,
                        const long      *countOfByte);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### countOfByte

The pointer to the number of bytes that will be returned. The maximum buffer length is 4 MB (4 194 304).

### Normal Return

A pointer to a buffer containing the requested data is returned.

### Error Return

An error return is indicated by NULL. When NULL is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- The caller must free the returned data buffer once the caller has stopped using the buffer. Enter a free function to free the buffer.
- This function leaves the cursor pointing at the next byte after the last returned byte. If the cursor is already pointing at the last byte in the collection, the TO2\_ERROR\_EODAD return code will be set.
- This function does not use TPF transaction services on behalf of the caller.

## T02\_nextRBAfor

### Examples

The following example shows how the next 10 bytes in the binary large object (BLOB) collection are returned and positions the cursor to point after the 10th byte.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_PID          cursor;
T02_ENV_PTR      env_ptr;
char             *buffer_ptr;
T02_ERR_CODE     err_code;      /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
:
/*****
/* Read the next 10 bytes in the BLOB and update the cursor. */
*****/
if ((buffer_ptr=(char *)T02_nextRBAfor(&cursor,
                                     env_ptr)) == NULL)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_ERROR_EODAD)
    {
        printf("T02_nextRBAfor failed!\n");
        process_error(env_ptr);
    }
    else
        printf("Cursor is at the end of the collection!\n");
    return(0);
}
else
    printf("T02_nextRBAfor successful!\n");
:
free(buffer_ptr);
```

### Related Information

- “T02\_next–Increment and Return the Next Element” on page 1159
- “T02\_peek–Return the Next Element with No Cursor Movement” on page 1168
- “T02\_previous–Return the Previous Element” on page 1172.

## TO2\_nextWithBuffer–Increment and Return the Next Element

This function returns the next element in the collection in the specified buffer and points the cursor at that element. The definition of the next element depends on the collection implementation.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_nextWithBuffer (const TO2_PID_PTR  cursorPidPtr,
                                   TO2_ENV_PTR    env_ptr,
                                   const long      *bufferLength,
                                   TO2_BUF_PTR    buffer);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### bufferLength

The pointer to the length of the specified buffer. The buffer should be large enough to hold the data to be returned plus 16 additional bytes for a header. The minimum buffer length is 20 bytes.

#### buffer

The pointer to the buffer where the element will be returned.

### Normal Return

For a normal return, this buffer contains a C header file followed by a copy of the data requested. The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859). If the length of the data is greater than the length of the buffer, only the amount of data that will fit is placed in the buffer. The length is returned in the buffer and is the actual length of the data element. The caller must check the returned length against the length of the supplied buffer to determine if the entire element has been returned.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_BUFFER_SIZE
TO2_ERROR_CURSOR
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- This function leaves the cursor pointing to the returned element. If the cursor is already pointing to the last element in the collection, the T02\_ERROR\_EODAD return code will be returned. The definition of next element depends on the collection implementation.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example reads the next item that the cursor is currently pointing to and returns it in the specified buffer.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */
#define DATA_SIZE 256          /* Size of data */

struct person_record
{
    char        name[50]
    char        address[50]
};
typedef struct person_record PERSON_RECORD;

T02_PID        cursor;
T02_ENV_PTR    env_ptr;
T02_BUF_PTR    buffer_ptr;
long           buffer_length;
PERSON_RECORD  *person_ptr;      /* Pointer to element data */
T02_ERR_CODE   err_code;        /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr;  /* T02 error code text pointer */
:
/*****
/* Read the next item that the cursor points to.
*****/
buffer_length = DATA_SIZE;
if ((buffer_ptr=malloc(buffer_length)) == NULL)
{
    printf("malloc() failed!\n");
    return(0);
}
if ((buffer_ptr = T02_nextWithBuffer(&cursor,
                                    env_ptr,
                                    &buffer_length,
                                    buffer_ptr)) == NULL)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_ERROR_EODAD)
    {
        printf("T02_nextWithBuffer failed!\n");
        process_error(env_ptr);
    }
}
else
    printf("T02_atCursor is at the end of the data!\n");
}
else
{
    person_ptr = buffer_ptr->data;
    if (buffer_length < (DATA_SIZE + sizeof(T02_BUF_HDR)))
        printf("Data returned did not fit in allocated buffer!\n");
    else
        printf("T02_atCursorWithBuffer successful!\n");
}
```

## **Related Information**

- “TO2\_next–Increment and Return the Next Element” on page 1159
- “TO2\_peek–Return the Next Element with No Cursor Movement” on page 1168
- “TO2\_previous–Return the Previous Element” on page 1172.

## T02\_peek—Return the Next Element with No Cursor Movement

This function returns the next element in the collection but does not update the cursor position. The definition of the next element depends on the collection implementation.

### Format

```
#include <c$to2.h>
T02_BUF_PTR T02_peek (const T02_PID_PTR cursorPidPtr,
                      T02_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

For a normal return, this buffer contains a C header file followed by a copy of the data requested. The normal return is a pointer (T02\_BUF\_PTR) to a structure (buffer) of type T02\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_CURSOR
T02_ERROR_EMPTY
T02_ERROR_ENV
T02_ERROR_EODAD
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

- The caller must free the returned buffer once the caller has stopped using the buffer. Enter a free function to free the buffer.
- This function does not change the position of the cursor.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example returns the next element in the collection that the cursor points to, but does not update the cursor position.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ERR_CODE    err_code;        /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr;   /* T02 error code text pointer */
T02_BUF_PTR     buffer_ptr;
T02_PID         cursor;
T02_ENV_PTR     env_ptr;
```

```

:
/*****
/* Read the next item that the cursor points to.
*****/
if ((buffer_ptr = T02_peek(&cursor,
                           env_ptr)) == NULL)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_ERROR_EODAD)
    {
        printf("T02_peek failed!\n");
        process_error(env_ptr);
    }
    else
        printf("T02_atCursor is at the end of the data!\n");
}
else
{
    person_ptr = buffer_ptr->data;
    if (buffer_length < (DATA_SIZE + sizeof(T02_BUF_HDR)))
        printf("Data returned did not fit in allocated buffer!\n");
    else
        printf("T02_peek successful!\n");
}

```

## Related Information

- “T02\_next–Increment and Return the Next Element” on page 1159
- “T02\_nextWithBuffer–Increment and Return the Next Element” on page 1165
- “T02\_previous–Return the Previous Element” on page 1172.

### TO2\_peekWithBuffer–Return the Next Element

This function returns the next element in the collection in the specified buffer but does not update the cursor position. The definition of the next element depends on the collection implementation.

#### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_peekWithBuffer (const TO2_PID_PTR cursorPidPtr,
                                  TO2_ENV_PTR env_ptr,
                                  const long *bufferLength,
                                  TO2_BUF_PTR buffer);
```

##### **cursorPidPtr**

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

##### **env\_ptr**

The pointer to the environment as returned by the TO2\_createEnv function.

##### **bufferLength**

The pointer to the length of the specified buffer. The buffer should be large enough to hold the data to be returned plus 16 additional bytes for a header. The minimum buffer length is 20 bytes.

##### **buffer**

The pointer to the buffer where the element will be returned.

#### Normal Return

For a normal return, this buffer contains a C header file followed by a copy of the data requested. The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859). If the length of the data is greater than the length of the buffer, only the amount of data that will fit is placed in the buffer. The length is returned in the buffer and is the actual length of the data element. The caller must check the returned length against the length of the supplied buffer to determine if the entire element has been returned.

#### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_BUFFER_SIZE
TO2_ERROR_CURSOR
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

#### Programming Considerations

- This function does not change the position of the cursor.
- This function does not use TPF transaction services on behalf of the caller.



## Examples

The following example reads the next element that the cursor is currently pointing to and returns it in the specified buffer without updating the positioning of the cursor.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions*/

T02_ERR_CODE    err_code;    /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
T02_BUF_PTR     buffer_ptr;
T02_PID         cursor;
T02_ENV_PTR     env_ptr;
long            buffer_len;
long            actual_buf_len;
:
:
/*****
/* Read the next item that the cursor points to. */
*****/
actual_buf_len = buffer_len;
if ((buffer_ptr = T02_peekWithBuffer(&cursor,
                                     env_ptr,
                                     &actual_buf_len,
                                     buffer_ptr)) == NULL)

{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_ERROR_EODAD)
    {
        printf("T02_peekWithBuffer failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_peekWithBuffer successful!\n");
}
}
```

## Related Information

- “T02\_next-Increment and Return the Next Element” on page 1159
- “T02\_nextWithBuffer-Increment and Return the Next Element” on page 1165
- “T02\_previous-Return the Previous Element” on page 1172.

## TO2\_previous–Return the Previous Element

This function returns the previous element in the collection and points the cursor at the element. The definition of the previous element depends on the collection implementation.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_previous (const TO2_PID_PTR pid_ptr,
                           TO2_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

### Normal Return

For a normal return, this buffer contains a C header file followed by a copy of the data requested. The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CURSOR
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_EODAD
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

## Programming Considerations

- The caller must free the returned buffer once the caller has stopped using the buffer. Enter a free function to free the buffer.
- This function leaves the cursor pointing at the returned element. If the cursor is already pointing at the first element in the collection, the TO2\_ERROR\_EODAD return code will be returned. The definition of previous element depends on the collection implementation.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example returns the previous element in the collection that the cursor points to and repositions the cursor to point to it.

```
#include <c$to2.h>                /* Needed for TO2 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

TO2_ERR_CODE    err_code;        /* TO2 error code value */
TO2_ERR_TEXT_PTR err_text_ptr;   /* TO2 error code text pointer */
```

```

T02_BUF_PTR    buffer_ptr;
T02_PID        cursor;
T02_ENV_PTR    env_ptr;
:
/*****
/* Read the previous item that the cursor points to.          */
*****/
if ((buffer_ptr = T02_previous(&cursor,
                             env_ptr)) == NULL)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_ERROR_EODAD)
    {
        printf("T02_previous failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_previous successful!\n");
}
}

```

## Related Information

- “TO2\_next–Increment and Return the Next Element” on page 1159
- “TO2\_nextWithBuffer–Increment and Return the Next Element” on page 1165
- “TO2\_previousWithBuffer–Return the Previous Element in the Specified Buffer” on page 1174
- “TO2\_remove–Remove the Element Pointed to by the Cursor” on page 1176.

## T02\_previousWithBuffer–Return the Previous Element in the Specified Buffer

This function returns the previous element in the collection in the specified buffer and points the cursor at the element. The definition of the previous element depends on the collection implementation.

### Format

```
#include <c$to2.h>
T02_BUF_PTR T02_previousWithBuffer (const T02_PID_PTR  pid_ptr,
                                     T02_ENV_PTR    env_ptr,
                                     const long      *bufferLength,
                                     T02_BUF_PTR    buffer);
```

#### pid\_ptr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### bufferLength

The pointer to the length of the specified buffer. The buffer should be large enough to hold the data to be returned plus 16 additional bytes for a header. The minimum buffer length is 20 bytes.

#### buffer

The pointer to the buffer where the element will be returned.

### Normal Return

For a normal return, this buffer contains a C header file followed by a copy of the data requested. The normal return is a pointer (T02\_BUF\_PTR) to a structure (buffer) of type T02\_BUF\_HDR (see “Type Definitions” on page 859). If the length of the data is greater than the length of the buffer, only the amount of data that will fit is placed in the buffer. The length is returned in the buffer and is the actual length of the data element. The caller must check the returned length against the length of the supplied buffer to determine if the entire element has been returned.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_BUFFER_SIZE
T02_ERROR_CURSOR
T02_ERROR_EMPTY
T02_ERROR_ENV
T02_ERROR_EODAD
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

- This function leaves the cursor pointing to the returned element. If the cursor is already pointing to the first element in the collection, the TO2\_ERROR\_EODAD return code will be returned. The definition of previous element depends on the collection implementation.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example reads the previous element that the cursor is currently pointing to, returns it in the specified buffer, and updates the cursor to point to it.

```
#include <c$to2.h>           /* Needed for TO2 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

TO2_ERR_CODE    err_code;    /* TO2 error code value */
TO2_ERR_TEXT_PTR err_text_ptr; /* TO2 error code text pointer */
TO2_BUF_PTR     buffer_ptr;
TO2_PID         cursor;
TO2_ENV_PTR     env_ptr;     /* Pointer to TO2 environment */
long            buffer_len;
long            actual_buf_len;
:
:
/*****
/* Read the previous item that the cursor points to and
/* return it in the specified buffer.
*****/
actual_buf_len = buffer_len;
if ((buffer_ptr = T02_atCursorWithBuffer(&cursor,
                                         env_ptr,
                                         &actual_buf_len,
                                         buffer_ptr)) == NULL)

{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code != T02_ERROR_EODAD)
    {
        printf("T02_previousWithBuffer failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_previousWithBuffer successful!\n");
}
}
else
    if (actual_buf_len < buffer_len)
    {
        printf("Data returned from cursor did not fit in allocated buffer.\n")
    }
}
```

## Related Information

- “TO2\_next-Increment and Return the Next Element” on page 1159
- “TO2\_nextWithBuffer-Increment and Return the Next Element” on page 1165
- “TO2\_previous-Return the Previous Element” on page 1172.

## T02\_remove—Remove the Element Pointed to by the Cursor

This function removes the element and the associated key of the element that the cursor is pointing to from the collection. It sets the position of the cursor at the following element or at the end of the collection if the element removed was the last element in the collection.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_remove (const T02_PID_PTR cursorPidPtr,
                 T02_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_CURSOR
T02_ERROR_EMPTY
T02_ERROR_ENV
T02_ERROR_EODAD
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

- This function leaves the cursor pointing to the next element in the collection or, if the removed element was the last element in the collection, the cursor will be pointed to the end of the collection. The definition of next element depends on the collection implementation.
- A commit scope will be created for the T02\_remove function. If the request is successful, the scope will be committed. If any error occurs while processing the T02\_remove request, the scope will be rolled back.

## Examples

The following example removes the element that the cursor is currently pointing to.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_PID          cursor;
```

```

TO2_ENV_PTR      env_ptr;
TO2_ERR_CODE     err_code;      /* T02 error code value      */
TO2_ERR_TEXT_PTR err_text_ptr;  /* T02 error code text pointer */
:
/*****
/* Remove the element currently pointed to by the cursor.
*****/
if (TO2_remove(&cursor,
               env_ptr) == T02_ERROR)
{
    printf("T02_remove failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_remove successful!\n");

```

## Related Information

- “TO2\_next–Increment and Return the Next Element” on page 1159
- “TO2\_nextWithBuffer–Increment and Return the Next Element” on page 1165
- “TO2\_previous–Return the Previous Element” on page 1172.

## T02\_reset—Reset Cursor to Point to First Element

This function resets the cursor to point to the first element in the collection.

### Format

```
#include <c$to2.h>
long T02_reset (const T02_PID_PTR cursorPidPtr,
                 T02_ENV_PTR env_ptr);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

- This function points the cursor to the first element in the collection.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example resets the cursor to point to the first element in the collection.

```
#include <c$to2.h>           /* Needed for T02 API Functions      */
#include <stdio.h>           /* APIs for standard I/O functions */
T02_PID cursor;
T02_ENV_PTR env_ptr; /* Pointer to T02 environment */
:
/*****
/* Point the cursor to the first element in the collection. */
*****/
if (T02_reset(&cursor,
              env_ptr) == T02_ERROR)
{
    printf("T02_reset failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_reset successful!\n");
```



## **Related Information**

- “TO2\_first–Point Cursor to First Element” on page 1139
- “TO2\_last–Point Cursor at Last Element” on page 1153.

## T02\_setKeyPath–Set a Cursor to Use a Specific Key Path

This function allows an application program to specify the key path a cursor will use to access a collection. When a cursor is created, the primary key path of the collection is used by default for searching and accessing data. This function allows an application program to override the default setting or any previous T02\_setKeyPath calls that were issued for this cursor.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_setKeyPath (const T02_PID_PTR cursorPid_ptr,
                    T02_ENV_PTR env_ptr,
                    const char *name);
```

#### cursorPid\_ptr

The pointer to the persistent identifier (PID) assigned to the target cursor.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### name

The pointer to a string that is the name of the key path that will be used by the cursor for accessing the collection.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_KEYPATH_BUILD_ACTIVE
T02_ERROR_LOCATOR_NOT_FOUND
T02_ERROR_METHOD
T02_ERROR_NOT_INIT
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

### Programming Considerations

- When a T02\_setKeyPath call is issued to a given cursor, the position of the cursor must be reestablished. If the T02\_setKeyPath request fails, the cursor will use the primary key path by default and the application must reestablish the position of the cursor before it can be used again.
- The application program can issue a T02\_setKeyPath call with the T02\_PRIME\_KEYPATH name to reset the cursor to use the primary key path.
- While the key path build process is in progress, the T02\_ERROR\_KEYPATH\_BUILD\_ACTIVE error code is returned. Until the build process ends, the key path is not usable.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example sets the cursor to use the specified key path.

```
#include <c$to2.h>           /* T02 API function prototypes */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;        /* Pointer to T02 Environment */
T02_PID keysetCursor;
char name="fieldA";         /* name of key path to remove */
:
:
if (T02_setKeyPath(&keysetCursor,
                  env_ptr,
                  name) == T02_ERROR)
{
    printf("T02_setKeyPath failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_setKeyPath was successful!\n");
}
:
:
/*****
/* The application must issue a positioning request such as T02_first. */
*****/
```

## Related Information

- “T02\_addKeyPath—Add a Key Path to a Collection” on page 890
- “T02\_getCurrentKey—Retrieve the Current Key” on page 1141
- “T02\_removeKeyPath—Remove a Key Path from a Collection” on page 1082.

## T02\_setPositionIndex–Point Cursor at Specified Position

This function points the cursor at the specified position. An integer index is required for arrays, binary large objects (BLOBs), logs, and sequence collections.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_setPositionIndex (const T02_PID_PTR  cursorPidPtr,
                           T02_ENV_PTR    env_ptr,
                           const long      *index);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### index

The pointer to the integer value that will be used as the current position value of the cursor.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_INDEX
T02_ERROR_METHOD
T02_ERROR_PID
T02_ERROR_ZERO_PID
```

## Programming Considerations

- This function positions the cursor at point to the specified element in the collection. If the element does not exist, the resulting action depends on the collection implementation.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example positions the cursor at position 10.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_PID      cursor;
T02_ENV_PTR  env_ptr;
long         index;
```

```

:
/*****
/* Point the cursor to the specified index position.          */
*****/
index = 10;
if (TO2_setPositionIndex(&cursor,
                        env_ptr,
                        &index) == T02_ERROR)
{
    printf("TO2_setPositionIndex failed!\n");
    process_error(env_ptr);
}
else
    printf("TO2_setPositionIndex successful!\n");

```

## Related Information

- “TO2\_locate—Locate Key and Point Cursor to Its Element” on page 1155
- “TO2\_setPositionValue—Point Cursor at Specified Element” on page 1184.

## T02\_setPositionValue—Point Cursor at Specified Element

This function points the cursor at the specified element. This function is not supported for arrays, binary large objects (BLOBs), logs, and sequence collections. For key sorted set, sorted bag, sorted key bag, and sorted set collections, a pointer to a key or sort field value is required. For bags and sets, a pointer to a value that will be used to access the collection is required. If there is more than one entry with the same value, the cursor will be positioned to the first entry found.

**Note:** This function does not support all collections. See Table 48 on page 1113 for collections that are supported for this function.

### Format

```
#include <c$to2.h>
long T02_setPositionValue (const T02_PID_PTR  cursorPidPtr,
                           T02_ENV_PTR      env_ptr,
                           const long       *valueL,
                           const void       *value);
```

#### cursorPidPtr

The pointer to the cursor persistent identifier (PID) created by one of the TPF collection support (TPFCS) create cursor application programming interfaces (APIs).

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### valueL

The pointer to an integer value that is the length of the passed value.

#### value

The pointer to a value that will be used to set the current position of the cursor.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_METHOD
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- This function positions the cursor to point to the specified element in the collection. If the element does not exist, the resulting action depends on the collection implementation.
- This function uses the current key path field to match the specified value (or the original key or sort field if no key path has been set).
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example points the cursor at the element matching the specified criteria.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

struct person_record
{
    char        name[20];
    char        address[60];
};
typedef struct person_record PERSON_RECORD;

T02_PID        cursor;
T02_ENV_PTR    env_ptr;
PERSON_RECORD  data;
long           dataLength;
:
:
/*****
/* Point the cursor to the element matching the key specified. */
*****/
dataLength = sizeof(PERSON_OBJ);
if (T02_setPositionValue(&cursor,
                        env_ptr,
                        &dataLength,
                        &data) == T02_ERROR)
{
    printf("T02_setPositionValue failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_setPositionValue successful!\n");
```

## Related Information

- “T02\_locate—Locate Key and Point Cursor to Its Element” on page 1155
- “T02\_setPositionIndex—Point Cursor at Specified Position” on page 1182.

**TO2\_setPositionValue**



---

## TPF Collection Support: Dictionary

This chapter provides information about TPF collection support (TPFCS) for dictionaries.

---

### Data Store Application Dictionary

Each data store contains a dictionary automatically created by TPFCS that is available for use by the application. This dictionary is known as the data store application dictionary, and is assigned the name DS\_USER\_DICT. This dictionary is accessed by establishing an environment for the target data store and using the T02\_...DSdict... type functions. The dictionary uses EBCDIC keys of 64 bytes with data elements of 1000 bytes.

One suggested use for the application dictionary is to place the persistent identifiers (PIDs) of data store anchor collections in the dictionary and assign a symbolic name to each one as the key.

**Note:** When PIDs are stored in collection elements, a recoup index must be established and associated with that collection.

The application dictionary of TPFDB, the base TPFCS data store, can be accessed with a special set of functions, T02\_...TPF.... This allows all applications access to a common dictionary without needing to establish an environment for a particular data store. A possible use for this dictionary is to associate applications with data stores.

**Note:** The data store system dictionaries, accessed with the T02\_...DSsystem... and T02\_...TPFsystem... type functions, are used by the TPFCS system and are not intended to be used by applications.

## TO2\_atDSdictKey–Retrieve the Element Using the Specified Key

This function locates the entry in the data store (DS) symbol dictionary represented by the specified key and returns its contents to the requester. If the key length is less than 64 bytes, the first entry with a key that matches the passed key string is returned.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_atDSdictKey (      TO2_ENV_PTR  env_ptr,
                                   const void    *key,
                                   const long     *keyLength);
```

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry.

#### keyLength

The pointer to a field containing the length of the key string.

### Normal Return

For a normal return, this buffer contains a C header file followed by a copy of the data requested. The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_LOCATOR_NOT_FOUND
```

## Programming Considerations

- The caller must free the returned buffer once the caller has finished using the buffer. Enter a free(\*buffer) function to free the buffer.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example retrieves an element from the symbol dictionary of the specified TPFCS data store.

```
#include <c$to2.h>           /* Needed for TO2 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

TO2_ERR_CODE  err_code;      /* TO2 error code value */
TO2_ERR_TEXT_PTR err_text_ptr; /* TO2 error code text pointer */

TO2_ENV_PTR  env_ptr;        /* Pointer to TO2 Environment */

TO2_BUF_PTR  buffer;         /* TO2 buffer pointer returned */

char key[64]
= "Dictionary.Symbol.Key";
```

```

long keyLength = sizeof(key);
:
if ((buffer = TO2_atDSdictKey(env_ptr,
                             key,
                             &keyLength)) == TO2_ERROR)
{
    printf("TO2_atDSdictKey failed!\n");
    err_code = TO2_getErrorCode(env_ptr);
    err_text_ptr = TO2_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
{
    printf("TO2_atDSdictKey successful !\n");
    free(buffer);          /* Release buffer when finished */
}

:

```

## Related Information

- “TO2\_atDSdictKeyPut—Replace the Element Using the Key” on page 1190
- “TO2\_atDSdictNewKeyPut—Store the Element Using the New Key” on page 1192
- “TO2\_removeDSdictKey—Remove the Element Using the Key” on page 1215.

## T02\_atDSdictKeyPut—Replace the Element Using the Key

This function locates the entry in the data store (DS) symbol dictionary and replaces its contents with the passed data values. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long T02_atDSdictKeyPut (      T02_ENV_PTR  env_ptr,
                              const void    *key,
                              const void    *data,
                              const long    *dataLength,
                              const long    *updateSeqCtr);
```

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **key**

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

#### **data**

The pointer to the data to place in the entry.

#### **dataLength**

The pointer to the length of the data to place in the entry.

#### **updateSeqCtr**

The pointer to an area where the update sequence counter from a T02\_at request has been stored. If this sequence counter value is not equal to the current update sequence counter value of the collection, the T02\_atPut request will not be run.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_LOCATOR_NOT_FOUND
T02_ERROR_SEQCTR
```

### Programming Considerations

A commit scope will be created for the T02\_atDSdictKeyPut request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_atDSdictKeyPut request, the scope will be rolled back.

### Examples

The following example updates an element in the TPFCS symbol dictionary for the specified data store.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                 /* APIs for standard I/O functions */
```

## TO2\_atDSdictKeyPut

```
T02_ENV_PTR      env_ptr;          /* Pointer to T02 Environment      */

char key[64]
= "Dictionary.Symbol.Key           ";

char data[]
= "This.is.some.data.for.the.T02.Symbol.Dictionary";

long dataLength = sizeof(data);
long UpdateSeqCtr = 0;

T02_ERR_CODE      err_code;          /* T02 error code value          */
T02_ERR_TEXT_PTR  err_text_ptr;      /* T02 error code text pointer   */
:
if (T02_atDSdictKeyPut(env_ptr,
                        key,
                        data,
                        &dataLength,
                        &updateSeqCtr) == T02_ERROR)
{
    printf("T02_atDSdictKeyPut failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
    printf("T02_atDSdictKeyPut successful!\n");
:
```

## Related Information

- “TO2\_atDSdictKey–Retrieve the Element Using the Specified Key” on page 1188
- “TO2\_atDSdictNewKeyPut–Store the Element Using the New Key” on page 1192
- “TO2\_removedSdictKey–Remove the Element Using the Key” on page 1215.

## T02\_atDSdictNewKeyPut—Store the Element Using the New Key

This function creates the entry in the data store (DS) symbol dictionary represented by the specified key and stores the passed data values as its contents. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long T02_atDSdictNewKeyPut (      T02_ENV_PTR  env_ptr,
                                const void    *key,
                                const void    *data,
                                const long    *dataLength);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

#### data

The pointer to the data to place in the entry.

#### dataLength

The pointer to the length of the data to place in the entry.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_LOCATOR_NOT_UNIQUE
```

### Programming Considerations

A commit scope will be created for the T02\_atDSdictNewKeyPut request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_atDSdictNewKeyPut request, the scope will be rolled back.

### Examples

The following example adds a new element to the TPFCS symbol dictionary for the specified data store.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */

char key[64]
= "Dictionary.Symbol.Key";

char data[]
= "This.is.some.data.for.the.T02.Class.Dictionary";

long dataLength = sizeof(data);
```

```

TO2_ERR_CODE    err_code;        /* T02 error code value          */
TO2_ERR_TEXT_PTR err_text_ptr;    /* T02 error code text pointer    */
:
if (TO2_atDSdictNewKeyPut(env_ptr,
                           key,
                           data,
                           &dataLength) == T02_ERROR)
{
    printf("T02_atDSdictNewKeyPut failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
{
    printf("T02_atDSdictNewKeyPut successful!\n");
    :
}

```

### Related Information

- “TO2\_atDSdictKey–Retrieve the Element Using the Specified Key” on page 1188
- “TO2\_atDSdictKeyPut–Replace the Element Using the Key” on page 1190
- “TO2\_removeDSdictKey–Remove the Element Using the Key” on page 1215.

## TO2\_atDSsystemKey–Retrieve the Element Using the Specified Key

This function locates the entry in the data store (DS) system dictionary represented by the specified key and returns its contents to the requester. If the key length is less than 64 bytes, the first entry with a key that matches the passed key string is returned.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_atDSsystemKey (      TO2_ENV_PTR env_ptr,
                                   const void    *key,
                                   const long     *keyLength);
```

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry.

#### keyLength

The pointer to a field containing the length of the key string.

### Normal Return

For a normal return, this buffer contains a C header file followed by a copy of the data requested. The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_LOCATOR_NOT_FOUND
```

### Programming Considerations

- The caller must free the returned buffer once the caller has finished using the buffer. Enter a free(\*buffer) function to free the buffer.
- This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example retrieves an element from the system dictionary of the specified TPFCS data store.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

TO2_ERR_CODE   err_code;    /* T02 error code value */
TO2_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */

TO2_ENV_PTR    env_ptr;    /* Pointer to T02 Environment */

TO2_BUF_PTR    buffer;     /* T02 buffer pointer returned */

char key[64]
= "Dictionary.Class.Key";
```



```

long keyLength = sizeof(key);
:
if ((buffer = TO2_atDSsystemKey(env_ptr,
                                key,
                                &keyLength)) == T02_ERROR)
{
    printf("TO2_atDSsystemKey failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
{
    printf("TO2_atDSsystemKey successful!\n");
    free(buffer);          /* Release buffer when finished */
}

:

```

## Related Information

- “TO2\_atDSsystemKeyPut—Replace the Element Using the Key” on page 1196
- “TO2\_atDSsystemNewKeyPut—Store the Element Using the New Key” on page 1198
- “TO2\_removeDSsystemKey—Remove the Element Using the Key” on page 1217.

## T02\_atDSsystemKeyPut—Replace the Element Using the Key

This function locates the entry in the data store (DS) system dictionary and replaces its contents with the passed data values. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long T02_atDSsystemKeyPut (      T02_ENV_PTR  env_ptr,
                                const void    *key,
                                const void    *data,
                                const long    *dataLength,
                                const long    *updateSeqCtr);
```

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **key**

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

#### **data**

The pointer to the data to place in the entry.

#### **dataLength**

The pointer to the length of the data to place in the entry.

#### **updateSeqCtr**

The pointer to an area where the update sequence counter from a T02\_at request has been stored. If this sequence counter value is not equal to the current update sequence counter value of the collection, the T02\_atPut request will not be run.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_LOCATOR_NOT_FOUND
T02_ERROR_SEQCTR
```

### Programming Considerations

A commit scope will be created for the T02\_atDSsystemKeyPut request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_atDSsystemKeyPut request, the scope will be rolled back.

### Examples

The following example updates an element in the TPFCS class dictionary for the specified data store.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */
```

## TO2\_atDSsystemKeyPut

```
TO2_ENV_PTR      env_ptr;          /* Pointer to T02 Environment      */

char key[64]
= "Dictionary.Class.Key";

char data[]
= "This.is.some.data.for.the.T02.Class.Dictionary";

long dataLength = sizeof(data);
long UpdateSeqCtr = 0;

TO2_ERR_CODE      err_code;          /* T02 error code value          */
TO2_ERR_TEXT_PTR  err_text_ptr;      /* T02 error code text pointer   */
:
if (TO2_atDSsystemKeyPut(env_ptr,
                        key,
                        data,
                        &dataLength,
                        &updateSeqCtr) == T02_ERROR)
{
    printf("TO2_atDSsystemKeyPut failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
    printf("TO2_atDSsystemKeyPut successful!\n");
:
```

## Related Information

- “TO2\_atDSsystemKey–Retrieve the Element Using the Specified Key” on page 1194
- “TO2\_atDSsystemNewKeyPut–Store the Element Using the New Key” on page 1198
- “TO2\_removeDSsystemKey–Remove the Element Using the Key” on page 1217.

## T02\_atDSsystemNewKeyPut–Store the Element Using the New Key

This function creates the entry in the data store (DS) system dictionary represented by the specified key and stores the passed data values as its contents. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long T02_atDSsystemNewKeyPut (      T02_ENV_PTR  env_ptr,
                                   const void    *key,
                                   const void    *data,
                                   const long     *dataLength);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

#### data

The pointer to the data to place in the entry.

#### dataLength

The pointer to the length of the data to place in the entry.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_UNIQUE
```

### Programming Considerations

A commit scope will be created for the T02\_atDSsystemNewKeyPut request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_atDSsystemNewKeyPut request, the scope will be rolled back.

### Examples

The following example adds a new element to the TPFCS class dictionary for the specified data store.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */

char key[64]
= "Dictionary.Class.Key";

char data[]
= "This.is.some.data.for.the.T02.Class.Dictionary";

long dataLength = sizeof(data);
```

```

TO2_ERR_CODE    err_code;          /* T02 error code value          */
TO2_ERR_TEXT_PTR err_text_ptr;     /* T02 error code text pointer   */
:
if (TO2_atDSsystemNewKeyPut(env_ptr,
                             key,
                             &data,
                             &datalength) == T02_ERROR)
{
    printf("T02_atDSsystemNewKeyPut failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
    printf("T02_atDSsystemNewKeyPut successful!\n");
:

```

### Related Information

- “TO2\_atDSsystemKey–Retrieve the Element Using the Specified Key” on page 1194
- “TO2\_atDSsystemKeyPut–Replace the Element Using the Key” on page 1196
- “TO2\_removedDSsystemKey–Remove the Element Using the Key” on page 1217.

## TO2\_atTPFKey–Retrieve the Element Using the Specified Key

This function locates the entry in the TPFDB data store (DS) represented by the specified key and returns its contents to the requester. If the key length is less than 64 bytes, the first entry with a key that matches the passed key string is returned.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_atTPFKey (      T02_ENV_PTR env_ptr,
                               const void   *key,
                               const long    *keyLength);
```

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **key**

The pointer to the EBCDIC key to use to locate the dictionary entry.

#### **keyLength**

The pointer to a field containing the length of the key string.

### Normal Return

The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_LOCATOR_NOT_FOUND
```

### Programming Considerations

- The caller must free the returned buffer once the caller has finished using the buffer. Enter a free(\*buffer) function to free the buffer.
- This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example reads an element from the TPF system dictionary.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */

T02_ERR_CODE   err_code;         /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr;   /* T02 error code text pointer */

T02_BUF_PTR buffer;             /* T02 buffer pointer returned */

u_char key[64]
= "TPF.Key.is.64.characters.long";

long keyLength = sizeof(key);
```

```

    :
    if ((buffer = T02_atTPFKey(env_ptr,
                              key,
                              &keyLength)) == T02_ERROR)
    {
        printf("atTPFKey failed!\n");
        err_code = T02_getErrorCode(env_ptr);
        err_text_ptr = T02_getErrorText(env_ptr, err_code);
        process_error(env_ptr);
        printf("err_text_ptr is %s\n", err_text_ptr);
    }
    else
        printf("atTPFKey is successful!\n");
    :
    free(buffer);                /* Release buffer when finished */

```

## Related Information

- “TO2\_atTPFKeyPut–Replace the Element Using the Specified Key” on page 1202
- “TO2\_atTPFNewKeyPut–Store the Element Using the New Key” on page 1204
- “TO2\_removeTPFKey–Remove the Element from the TPF Dictionary” on page 1219.

## T02\_atTPFKeyPut–Replace the Element Using the Specified Key

This function locates the entry represented by the specified key and replaces its contents with the passed data values. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long T02_atTPFKeyPut (      T02_ENV_PTR  env_ptr,
                           const void    *key,
                           const long     *dataLength,
                           const void     *data,
                           const long     *updateSeqCtr);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

#### dataLength

The pointer to the length of the data to place in the entry.

#### data

The pointer to the data to place in the entry.

#### updateSeqCtr

The pointer to an area where the update sequence counter from a T02\_at request has been stored. If this sequence counter value is not equal to the current update sequence counter value of the collection, the T02\_atPut request will not be run.

### Normal Return

The normal return is a positive value. The entry represented by the specified key is updated in the dictionary.

### Error Return

An error return is indicated by a zero. When zero is returned, the T02\_getErrorCode function can be used to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_LOCATOR_NOT_FOUND
T02_ERROR_SEQCTR
```

### Programming Considerations

A commit scope will be created for the T02\_atTPFKeyPut request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_atTPFKeyPut request, the scope will be rolled back.

### Examples

The following example updates an element in the TPF system dictionary.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                 /* APIs for standard I/O functions */
```



```

TO2_ERR_CODE    err_code;        /* T02 error code value          */
TO2_ERR_TEXT_PTR err_text_ptr;    /* T02 error code text pointer    */

u_char key[64]
= "TPF.Key.is.64.characters.long";

u_char data[]
= "This.is.some.data.for.the.TPF.Dictionary";

long dataLength;
long UpdateSeqCtr = 0;
:
:
datalength = sizeof(data);

if (TO2_atTPFKeyPut(env_ptr,
                    key,
                    &dataLength,
                    data,
                    &updateSeqCtr) == T02_ERROR)
{
    printf("atTPFKeyPut failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
    printf("atTPFKeyPut is successful!\n");
:
:

```

## Related Information

- “TO2\_atTPFKey–Retrieve the Element Using the Specified Key” on page 1200
- “TO2\_atTPFNewKeyPut–Store the Element Using the New Key” on page 1204
- “TO2\_removeTPFKey–Remove the Element from the TPF Dictionary” on page 1219.

## T02\_atTPFNewKeyPut–Store the Element Using the New Key

This function creates the entry represented by the specified key and stores the passed data values as its contents. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long T02_atTPFNewKeyPut (      T02_ENV_PTR  env_ptr,
                              const void    *key,
                              const long    *dataLength,
                              const void    *data);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

#### dataLength

The pointer to the length of the data to place in the entry.

#### data

The pointer to the data to place in the entry.

### Normal Return

The normal return is a positive value. A new entry is added to the dictionary represented by the specified key.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_UNIQUE
```

### Programming Considerations

A commit scope will be created for the T02\_atTPFNewKeyPut request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_atTPFNewKeyPut request, the scope will be rolled back.

### Examples

The following example adds a new element to the TPF system dictionary.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */
T02_ERR_CODE   err_code;         /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr;   /* T02 error code text pointer */

char key[64]
= "TPF.Key.is.64.characters.long";

u_char data[]
= "This.is.some.data.for.the.TPF.Dictionary";
```

```

long dataLength = sizeof(data);
:
if (TO2_atTPFNewKeyPut(env_ptr,
                        key,
                        &dataLength,
                        data) == TO2_ERROR)
{
    printf("atTPFNewKeyPut failed!\n");
    err_code = TO2_getErrorCode(env_ptr);
    err_text_ptr = TO2_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
    printf("atTPFNewKeyPut is successful!\n");
:

```

## Related Information

- “TO2\_atTPFKey–Retrieve the Element Using the Specified Key” on page 1200
- “TO2\_atTPFKeyPut–Replace the Element Using the Specified Key” on page 1202
- “TO2\_removeTPFKey–Remove the Element from the TPF Dictionary” on page 1219.

## TO2\_atTPFsystemKey–Retrieve the Element Using the Specified Key

This function locates the entry represented by the specified key and returns its contents to the requester. If the key length is less than 64 bytes, the first entry with a key that matches the passed key string is returned.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_atTPFsystemKey (      TO2_ENV_PTR  env_ptr,
                                     const void    *key,
                                     const long     *keyLength);
```

#### **env\_ptr**

The pointer to the environment as returned by the TO2\_createEnv function.

#### **key**

The pointer to the EBCDIC key to use to locate the dictionary entry.

#### **keyLength**

The pointer to a field containing the length of the key string.

### Normal Return

The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_LOCATOR_NOT_FOUND
```

### Programming Considerations

- The caller must free the returned buffer once the caller has finished using the buffer. Enter a free(\*buffer) function to free the buffer.
- This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example reads an element from the TPF system dictionary.

```
#include <c$to2.h>           /* Needed for TO2 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

TO2_ENV_PTR  env_ptr;       /* Pointer to TO2 Environment */

TO2_ERR_CODE err_code;      /* TO2 error code value */
TO2_ERR_TEXT_PTR err_text_ptr; /* TO2 error code text pointer */

TO2_BUF_PTR buffer;         /* TO2 buffer pointer returned */

u_char key[64]
= "TPF.Key.is.64.characters.long";

long keyLength = sizeof(key);
```

```

    :
    if ((buffer = TO2_atTPFsystemKey(env_ptr,
                                    key,
                                    &keyLength)) == T02_ERROR)
    {
        printf("atTPFsystemKey failed!\n");
        err_code = T02_getErrorCode(env_ptr);
        err_text_ptr = T02_getErrorText(env_ptr, err_code);
        printf("err_text_ptr is %s\n", err_text_ptr);
    }
    else
        printf("atTPFsystemKey is successful!\n");
    :
    free(buffer);          /* Release buffer when finished    */

```

## Related Information

- “TO2\_atTPFsystemKeyPut—Replace the Element Using the Specified Key” on page 1208
- “TO2\_atTPFsystemNewKeyPut—Store the Element Using the New Key” on page 1210
- “TO2\_removeTPFsystemKey—Remove the Element from the TPF System Dictionary” on page 1221.

## T02\_atTPFsystemKeyPut—Replace the Element Using the Specified Key

This function locates the entry represented by the specified key and replaces its contents with the passed data values. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long T02_atTPFsystemKeyPut (      T02_ENV_PTR  env_ptr,
                                const void      *key,
                                const long      *dataLength,
                                const void      *data,
                                const long      *updateSeqCtr);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

#### dataLength

The pointer to the length of the data to place in the entry.

#### data

The pointer to the data to place in the entry.

#### updateSeqCtr

The pointer to an area where the update sequence counter from a T02\_at request has been stored. If this sequence counter value is not equal to the current update sequence counter value of the collection, the T02\_atPut request will not be run.

### Normal Return

The normal return is a positive value. The entry represented by the specified key is updated in the dictionary.

### Error Return

An error return is indicated by a zero. When zero is returned, the T02\_getErrorCode function can be used to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_LOCATOR_NOT_FOUND
T02_ERROR_SEQCTR
```

### Programming Considerations

A commit scope will be created for the T02\_atTPFsystemKeyPut request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_atTPFsystemKeyPut request, the scope will be rolled back.

### Examples

The following example updates an element in the TPF system dictionary.

## TO2\_atTPFsystemKeyPut

```
#include <c$to2.h>          /* Needed for T02 API Functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ERR_CODE    err_code;    /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */

u_char key[64]
= "TPF.Key.is.64.characters.long";

u_char data[]
= "This.is.some.data.for.the.TPF.Dictionary";

long dataLength;
long UpdateSeqCtr = 0;
:
:
dataLength = sizeof(data);

if (TO2_atTPFsystemKeyPut(env_ptr,
                          key,
                          &dataLength,
                          data,
                          &updateSeqCtr) == T02_ERROR)
{
    printf("atTPFsystemKeyPut failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
{
    printf("atTPFsystemKeyPut is successful!\n");
    :
    :
}
```

## Related Information

- “TO2\_atTPFsystemKey–Retrieve the Element Using the Specified Key” on page 1206
- “TO2\_atTPFsystemNewKeyPut–Store the Element Using the New Key” on page 1210
- “TO2\_removeTPFsystemKey–Remove the Element from the TPF System Dictionary” on page 1221.

## T02\_atTPFsystemNewKeyPut–Store the Element Using the New Key

This function creates the entry represented by the specified key and stores the passed data values as its contents. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long T02_atTPFsystemNewKeyPut (      T02_ENV_PTR env_ptr,
                                   const void    *key,
                                   const long     *dataLength,
                                   const void     *data);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

#### dataLength

The pointer to the length of the data to place in the entry.

#### data

The pointer to the data to place in the entry.

### Normal Return

The normal return is a positive value. A new entry is added to the dictionary represented by the specified key.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_UNIQUE
```

### Programming Considerations

A commit scope will be created for the T02\_atTPFsystemNewKeyPut request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_atTPFsystemNewKeyPut request, the scope will be rolled back.

### Examples

The following example adds a new element to the TPF system dictionary.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */
T02_ERR_CODE   err_code;         /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr;   /* T02 error code text pointer */

char key[64]
= "TPF.Key.is.64.characters.long";

u_char data[]
= "This.is.some.data.for.the.TPF.Dictionary";
```



```

long dataLength = sizeof(data);
:
if (TO2_atTPFsystemNewKeyPut(env_ptr,
                             key,
                             &dataLength,
                             data) == T02_ERROR)
{
    printf("atTPFsystemNewKeyPut failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
    printf("atTPFsystemNewKeyPut is successful!\n");
:

```

## Related Information

- “TO2\_atTPFsystemKey–Retrieve the Element Using the Specified Key” on page 1206
- “TO2\_atTPFsystemKeyPut–Replace the Element Using the Specified Key” on page 1208
- “TO2\_removeTPFsystemKey–Remove the Element from the TPF System Dictionary” on page 1221.

## T02\_getDSdictPID—Get the Dictionary PID of a Data Store

This function returns the persistent identifier (PID) of the symbolic dictionary for the data store (DS) specified in the passed environment to the specified field.

### Format

```
#include <c$to2.h>
long T02_getDSdictPID (T02_PID_PTR  pid_ptr,
                      T02_ENV_PTR  env_ptr);
```

#### pid\_ptr

The pointer to a field where the dictionary PID of the data store will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value and the specified T02\_PID\_PTR field will contain the PID of the data store dictionary.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_NOT_INIT
```

### Programming Considerations

None.

### Examples

The following example copies the PID of the data store dictionary for data store TEST1\_DS.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */
T02_ENV_PTR  env_ptr;            /* Pointer to T02 environment */
T02_PID      dictionaryPID;      /* return area for PID */
long         userTkn = 0;
char         ApplicationName[33]="ApplicationName1";
char         DSname[]="TEST1_DS"; /* data store name blank padded */
T02_ERR_CODE to2_rc=1;          /* return code receiver */
T02_ERR_TEXT_PTR err_textPtr;    /* T02 error code text pointer */
:
{
if ((to2_rc = T02_createEnv(&env_ptr,
                          &userTkn,
                          ApplicationName,
                          DSname)) == T02_ERROR)
{
to2_rc = T02_getErrorCode(env_ptr);
err_textPtr = T02_getErrorText(env_ptr, to2_rc);
printf ("T02_createEnv failed, error code - %d\n ",
        to2_rc);
printf ("T02 Error Text is %s\n ", err_textPtr);
}
else
```

```
{
if ((to2_rc = TO2_getDSdictPID(&dictionaryPID,
                             env_ptr)) == TO2_ERROR)
{
    to2_rc = TO2_getErrorCode(env_ptr);
    err_textPtr = TO2_getErrorText(env_ptr, to2_rc);
    printf ("TO2_getDSdictPID failed, error code - %d\n ", to2_rc);
    printf ("TO2 Error Text is %s\n ", err_textPtr);
}
else
    printf("TO2 get data store's dictionary's PID successful\n");
}
}
```

## Related Information

"TO2\_getTPFDictPID—Get the PID of the TPF Dictionary" on page 1214.

## T02\_getTPFDictPID—Get the PID of the TPF Dictionary

This function returns the persistent identifier (PID) of the TPF dictionary to the specified field.

### Format

```
#include <c$to2.h>
long T02_getTPFDictPID (T02_PID_PTR  pid_ptr,
                       T02_ENV_PTR  env_ptr);
```

#### pid\_ptr

The pointer to a field where the PID of the TPF dictionary will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value and the specified T02\_PID\_PTR field will contain the PID of the dictionary.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_NOT_INIT
```

### Programming Considerations

None.

### Examples

The following example copies the PID of the TPF dictionary.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */
T02_ENV_PTR  env_ptr;       /* Pointer to T02 environment */
T02_PID      dictionaryPID; /* return area for PID */

T02_ERR_CODE to2_rc=1;      /* return code receiver */
T02_ERR_TEXT_PTR err_textPtr; /* T02 error code text pointer */
:
{
if ((to2_rc = T02_getTPFDictPID(&dictionaryPID,
                               env_ptr)) == T02_ERROR)
{
to2_rc = T02_getErrorCode(env_ptr);
err_textPtr = T02_getErrorText(env_ptr, to2_rc);
printf ("T02_getTPFDictPID failed, error code - %d\n ", to2_rc);
printf ("T02 Error Text is %s\n ", err_textPtr);
}
else
printf("T02 get TPF dictionary's PID successful\n");
}
```

### Related Information

“T02\_getDSdictPID—Get the Dictionary PID of a Data Store” on page 1212.

## T02\_removeDSdictKey—Remove the Element Using the Key

This function locates the entry in the data store (DS) symbol dictionary and removes it from the dictionary. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long T02_removeDSdictKey (      T02_ENV_PTR  env_ptr,
                              const void    *key);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_LOCATOR_NOT_FOUND
```

### Programming Considerations

A commit scope will be created for the T02\_removeDSdictKey request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_removeDSdictKey request, the scope will be rolled back.

### Examples

The following example removes an element from the TPFCS symbol dictionary for the specified data store.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* Pointer to T02 Environment */

char key[64]
= "Dictionary.Symbol.Key";

T02_ERR_CODE    err_code;   /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
:
:
if (T02_removeDSdictKey(env_ptr,
                        key) == T02_ERROR)
{
    printf("T02_removeDSdictKey failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
```

## TO2\_removeDSdictKey

```
printf("TO2_removeDSdictKey successful!\n");  
:
```

## Related Information

- “TO2\_atDSdictKey—Retrieve the Element Using the Specified Key” on page 1188
- “TO2\_atDSdictKeyPut—Replace the Element Using the Key” on page 1190
- “TO2\_atDSdictNewKeyPut—Store the Element Using the New Key” on page 1192.

## T02\_removeDSsystemKey—Remove the Element Using the Key

This function locates the entry in the data store (DS) system dictionary and removes it from the dictionary. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long T02_removeDSsystemKey (      T02_ENV_PTR env_ptr,
                                const void *key);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_LOCATOR_NOT_FOUND
```

### Programming Considerations

A commit scope will be created for the T02\_removeDSsystemKey request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_removeDSsystemKey request, the scope will be rolled back.

### Examples

The following example removes an element from the TPFCS class dictionary for the specified data store.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                 /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */

char key[64]
= "Dictionary.System.Key";

T02_ERR_CODE   err_code;         /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr;   /* T02 error code text pointer */
:
:
if (T02_removeDSsystemKey(env_ptr,
                          key) == T02_ERROR)
{
    printf("T02_removeDSsystemKey failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
```

## TO2\_removeDSsystemKey

```
printf("TO2_removeDSsystemKey successful!\n");  
:
```

## Related Information

- “TO2\_atDSsystemKey–Retrieve the Element Using the Specified Key” on page 1194
- “TO2\_atDSsystemKeyPut–Replace the Element Using the Key” on page 1196
- “TO2\_atDSsystemNewKeyPut–Store the Element Using the New Key” on page 1198.



## T02\_removeTPFKey—Remove the Element from the TPF Dictionary

This function locates the entry represented by the specified key and removes the entry from the dictionary. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long T02_removeTPFKey (      T02_ENV_PTR env_ptr,
                           const void    *key);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

### Normal Return

The normal return is a positive value. The dictionary entry represented by the specified key is deleted.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_LOCATOR_NOT_FOUND
```

### Programming Considerations

A commit scope will be created for the T02\_removeTPFKey request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_removeTPFKey request, the scope will be rolled back.

### Examples

The following example removes an element from the TPF system dictionary.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;     /* Pointer to T02 Environment */
T02_ERR_CODE   err_code;    /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */

u_char key[64]
= "TPF.Key.is.64.characters.long";
:
:
if (T02_removeTPFKey(env_ptr,
                    key) == T02_ERROR)
{
    printf("removeTPFKey failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
```

## TO2\_removeTPFKey

```
printf("removeTPFkey is successful!\n");  
:
```

## Related Information

- “TO2\_atTPFKey–Retrieve the Element Using the Specified Key” on page 1200
- “TO2\_atTPFKeyPut–Replace the Element Using the Specified Key” on page 1202
- “TO2\_atTPFNewKeyPut–Store the Element Using the New Key” on page 1204.

## TO2\_removeTPFsystemKey—Remove the Element from the TPF System Dictionary

This function locates the entry represented by the specified key and removes the entry from the dictionary. The passed key is assumed to be 64 bytes in length.

### Format

```
#include <c$to2.h>
long TO2_removeTPFsystemKey (      T02_ENV_PTR env_ptr,
                                const void    *key);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key is assumed to be 64 bytes in length.

### Normal Return

The normal return is a positive value. The dictionary entry represented by the specified key is deleted.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_LOCATOR_NOT_FOUND
```

### Programming Considerations

A commit scope will be created for the T02\_removeTPFsystemKey request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_removeTPFsystemKey request, the scope will be rolled back.

### Examples

The following example removes an element from the TPF system dictionary.

```
#include <c$to2.h>          /* Needed for T02 API Functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* Pointer to T02 Environment */
T02_ERR_CODE   err_code;   /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */

u_char key[64]
= "TPF.Key.is.64.characters.long";
:
:
if (TO2_removeTPFsystemKey(env_ptr,
                          key) == T02_ERROR)
{
    printf("removeTPFsystemKey failed!\n");
    err_code = T02_getErrorCode(env_ptr);
    err_text_ptr = T02_getErrorText(env_ptr, err_code);
    printf("err_text_ptr is %s\n", err_text_ptr);
}
else
```

## TO2\_removeTPFsystemKey

```
printf("removeTPFsystemkey is successful!\n");  
:
```

## Related Information

- “TO2\_atTPFsystemKey–Retrieve the Element Using the Specified Key” on page 1206
- “TO2\_atTPFsystemKeyPut–Replace the Element Using the Specified Key” on page 1208
- “TO2\_atTPFsystemNewKeyPut–Store the Element Using the New Key” on page 1210.

---

## TPF Collection Support: Browser

This chapter provides information about browser support, which allows classes, methods, and collections of TPFCS to be located, interrogated, verified, displayed, and dumped. This support is provided by using the ZBROW commands and TPFCS function calls. For more information about the ZBROW commands, see *TPF Operations*.

## TO2\_atBrowseKey–Retrieve the Element Using the Specified Key from the Browser Dictionary

This function locates the entry represented by the specified key and returns its contents to the requester. If the key length is less than 64 bytes, the first entry with a key that matches the passed key string will be returned.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_atBrowseKey (      TO2_ENV_PTR env_ptr,
                                   const void    *key,
                                   const long     *keyLength);
```

#### **env\_ptr**

The pointer to the environment as returned by the TO2\_createEnv function.

#### **key**

The pointer to the EBCDIC key to use to locate the dictionary entry.

#### **keyLength**

The pointer to a field containing the length of the key.

### Normal Return

The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_EMPTY
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_LGH
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

None.

### Examples

The following example retrieves the element from the browser dictionary using a specified key.

```
#include <c$to2.h>           /* Needed for TO2 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */
#define UNDEF -1             /* Error return indication */

long      err_code;
char      key[64];
TO2_BUF_PTR buf_ptr;        /* TO2 buffer pointer returned */
TO2_ENV_PTR env_ptr;        /* Pointer to TO2 environment */
long      keyLength;
```

```

        :
buf_ptr = TO2_atBrowseKey(env_ptr,
                          key,
                          &keyLength);

if (buf_ptr == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code == T02_ERROR_LOCATOR_NOT_FOUND) return UNDEF;

    {
        printf("TO2_atBrowseKey failed!\n");
        process_error(env_ptr);
    }

else
{
    printf("TO2_atBrowseKey successful!\n");
}
}

```

## Related Information

- “TO2\_atBrowseKeyPut—Replace the Element Using the Specified Key from the Browser Dictionary” on page 1226
- “TO2\_atBrowseNewKeyPut—Store the Element Using the Specified Key in the Browser Dictionary” on page 1228
- “TO2\_removeBrowseKey—Remove the Element Using the Specified Key” on page 1332.

## T02\_atBrowseKeyPut—Replace the Element Using the Specified Key from the Browser Dictionary

This function locates the entry represented by the specified key and replaces its contents with the passed data values. The passed key string is assumed to be 64 bytes long.

### Format

```
#include <c$to2.h>
long T02_atBrowseKeyPut (      T02_ENV_PTR  env_ptr,
                              const void    *key,
                              const long    *dataLength,
                              const void    *data,
                              const long    *updateSeqCtr);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key string is assumed to be 64 bytes long.

#### dataLength

The pointer to the length of the data to place in the entry.

#### data

The pointer to the data to place in the entry.

#### updateSeqCtr

The pointer to an area where the update sequence counter from a T02\_atBrowseKey request has been stored. The T02\_atBrowseKey request was previously done to retrieve the data that is now being replaced. If this sequence counter value is not equal to the current update sequence counter value of the collection, the T02\_atBrowseKeyPut request will not be run.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_SEQCTR
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```



## Programming Considerations

A commit scope will be created for the TO2\_atBrowseKeyPut function. If the request is successful, the scope will be committed. If an error occurs while processing the TO2\_atBrowseKeyPut request, the scope will be rolled back.

## Examples

The following example replaces the data in the dictionary with the passed data values.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */
#define UNDEF -1            /* Error return indication */

long      err_code;
char      *key;
char      *data;
T02_ENV_PTR env_ptr;        /* Pointer to T02 environment */
long      updateSeqCtr;
long      dataLength;
:
:
err_code = T02_atBrowseKeyPut (env_ptr,
                              key,
                              &dataLength,
                              data,
                              &updateSeqCtr);

if (err_code == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code == T02_ERROR_LOCATOR_NOT_FOUND) return UNDEF;
    {
        printf("T02_atBrowseKeyPut failed!\n");
        process_error(env_ptr);
    }
}

else
{
    printf("T02_atBrowseKeyPut successful!\n");
}
}
```

## Related Information

- “TO2\_atBrowseKey–Retrieve the Element Using the Specified Key from the Browser Dictionary” on page 1224
- “TO2\_atBrowseNewKeyPut–Store the Element Using the Specified Key in the Browser Dictionary” on page 1228
- “TO2\_getBrowseDictPID–Get the Browser Dictionary PID” on page 1254
- “TO2\_removeBrowseKey–Remove the Element Using the Specified Key” on page 1332.

## T02\_atBrowseNewKeyPut—Store the Element Using the Specified Key in the Browser Dictionary

This function creates the entry represented by the specified key and stores the passed data values as its contents. The passed key string is must be 64 bytes long.

### Format

```
#include <c$to2.h>
long T02_atBrowseNewKeyPut (      T02_ENV_PTR  env_ptr,
                                const void      *key,
                                const void      *data,
                                const long      *dataLength);
```

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **key**

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key string must be 64 bytes long.

#### **data**

The pointer to the data to place in the entry.

#### **dataLength**

The pointer to the length of the data to place in the entry.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_UNIQUE
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_UPDATE_NOT_ALLOWED
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- A commit scope will be created for the T02\_atBrowseNewKeyPut function. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_atBrowseNewKeyPut request, the scope will be rolled back.
- Duplicate keys are not allowed.

### Examples

The following example adds data to the dictionary with the key specified.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

long          err_code;
```

```

char      *key;
char      *data;
T02_ENV_PTR env_ptr;      /* Pointer to T02 environment */
long      dataLength;
:
err_code = T02_atBrowseNewKeyPut (env_ptr,
                                key,
                                &dataLength,
                                data);

if (err_code == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);

    {
        printf("T02_atBrowseNewKeyPut failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_atBrowseNewKeyPut successful!\n");
}
}

```

## Related Information

- “TO2\_atBrowseKey–Retrieve the Element Using the Specified Key from the Browser Dictionary” on page 1224
- “TO2\_atBrowseKeyPut–Replace the Element Using the Specified Key from the Browser Dictionary” on page 1226
- “TO2\_getBrowseDictPID–Get the Browser Dictionary PID” on page 1254
- “TO2\_removeBrowseKey–Remove the Element Using the Specified Key” on page 1332.

## T02\_changeDD—Change a Data Definition

This function changes a data definition (DD).

### Format

```
#include <c$to2.h>
long T02_changeDD (      T02_ENV_PTR    env_ptr,
                        const char      DDname[T02_MAX_DDNAME],
                        const char      dsname[T02_MAX_DSNAME],
                        const T02_OPTION_PTR optionListPtr);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### DDname

The pointer to a character string, which is the name of the DD being changed. The character string is assumed to be 32 characters in length. The name should be left-justified and padded with blanks (X'40') on the right if it less than 32 characters.

#### dsname

The pointer to a character string, which is the data store (DS) name associated with the DD being changed. The character string must be 8 characters or greater in length, left-justified, and padded with blanks (X'40') on the right.

#### optionListPtr

This is the pointer to the option list.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DD_NOT_FOUND
TO2_ERROR_LOCATOR_NOT_FOUND
```

## Programming Considerations

A commit scope will be created for the T02\_changeDD request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_changeDD request, the scope will be rolled back.

### Examples

The following updates the data definition name (DDname) with the string passed in dsname.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */
#define UNDEF -1           /* Error return indication */

long      err_code;
char      DDname[32];
char      dsname[8];
T02_OPTION_PTR optionListPtr
T02_ENV_PTR env_ptr;      /* Pointer to T02 environment */
```

```

:
TO2_changeDD(env_ptr,
             DDname,
             dsname,
             optionListPtr);
if (err_code == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    if (err_code == T02_ERROR_DD_NOT_FOUND) return UNDEF;
    {
        printf("T02_changeDD failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_changeDD successful!\n");
}
}

```

## Related Information

- “TO2\_createDD—Create a Data Definition” on page 1238
- “TO2\_deleteDD—Delete a Data Definition” on page 1250
- “TO2\_getDDAttributes—Get the Current DD Attribute Values” on page 1274
- “TO2\_getListDDnames—Retrieve the Defined Data Definition Names” on page 1283.

## T02\_changeDS—Change the Attributes of a Data Store

This function changes data store (DS) attributes using a passed option list.

### Format

```
#include <c$to2.h>
long T02_changeDS (      T02_ENV_PTR    env_ptr,
                        const char      dsname,
                        const T02_OPTION_PTR optionListPtr);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### dsname

The pointer to a character string, which is the name of the DS being changed. The character string must be 8 characters or greater in length, left-justified, and padded with blanks (X'40') on the right.

#### optionListPtr

The pointer to a returned option list from a T02\_createOptionList call.

**Note:** The T02\_DS... options are used with the DS option list (T02\_OPTION\_LIST\_DS).

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

## Programming Considerations

A commit scope will be created for the T02\_changeDS request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_changeDS request, the scope will be rolled back.

## Examples

The following example creates an inventory collection if one does not exist. This example also changes a specified data store to use immediate deletes.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;             /* Pointer to T02 environment */
char      dsname[]="TESTDS1 ";  /* data store to change */
T02_OPTION_PTR optionListPtr;

                                /* invoke T02 to create option list */
:
optionListPtr=T02_createOptionList(env_ptr,
                                T02_OPTION_LIST_DS, /* create options */
                                T02_DS_INVENTORY, /* create a PID */
                                /* inventory collection */
                                T02_DS_DELETE_IMMED, /* process collection */
                                /* deletions immediately*/
                                T02_OPTION_LIST_END); /* end of options */
```

```
if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (TO2_changeDS(env_ptr,
                 dsname,
                 optionListPtr) == T02_ERROR)
{
    printf("TO2_changeDS failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_changeDS successful!\n");
}
```

## Related Information

- “TO2\_createDS—Create a Data Store” on page 1240
- “TO2\_getDSAttributes—Get the Attributes of the Data Store” on page 1278
- “TO2\_getDSnameForPID—Determine the DS Name for a PID” on page 1280
- “TO2\_getListDSnames—Retrieve the Defined Data Store Names” on page 1287.

## T02\_convertClassName—Convert EBCDIC Class Name to Class Index

This function converts an EBCDIC class name to its corresponding class index.

### Format

```
#include <c$to2.h>
long T02_convertClassName (      T02_ENV_PTR env_ptr,
                                const long   *classNameLength,
                                const char    className[T02_MAX_CLASS_NAME]);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### classNameLength

The pointer to the length of the input class name.

#### className

The pointer to a character string, which is the name of the class that will be converted.

### Normal Return

The normal return is the class index of the class name passed as a parameter.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

T02\_ERROR\_CLASS\_NOTFOUND

### Programming Considerations

None.

### Examples

The following example converts the class name provided into a class index.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;      /* Pointer to T02 environment */
char      *class_name;
long      classNameLength;
:
:
if (T02_convertClassName(env_ptr,
                        &classNameLength,
                        class_name) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);

    {
        printf("T02_convertClassName failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_convertClassName successful!\n");
}
}
```



## **Related Information**

"TO2\_getClassNames—Convert EBCDIC Class Name to Index" on page 1262.

## T02\_convertMethodName—Convert Method Name to Entry Address

This function causes TPF collection support (TPFCS) to locate the specific method definition in storage and return its entry point address to the caller.

### Format

```
#include <c$to2.h>
long T02_convertMethodName (T02_ENV_PTR env_ptr,
                           long         *class_namelen,
                           char         class_name[T02_MAX_CLASS_NAME],
                           long         *method_namelen,
                           char         method_name[64]);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### class\_namelen

A pointer to a field that contains the length of the class name.

#### class\_name

A pointer to a field that contains the class name.

#### method\_namelen

A pointer to a field that contains the length of the method name.

#### method\_name

A pointer to a field that contains the method name.

### Normal Return

TPFCS returns a positive address value as the return value, which is the entry point address of the method in memory.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_CLASS_NOTFOUND
T02_ERROR_ENV
T02_ERROR_METHOD_NOTFOUND
T02_ERROR_NOT_INIT
```

### Programming Considerations

None.

### Examples

The following example locates the method address for class methodName new.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;             /* Pointer to T02 environment */

char className[]="OBJECT";       /* class name */
long classNameLength=(sizeof "OBJECT")-1; /* length of class name */

char methodName[]="new";         /* method name */
long methodNameLength=(sizeof "new")-1; /* length of method name */
```

## TO2\_convertMethodName

```

T02_ERR_CODE    to2_rc=1;          /* method name          */
T02_ERR_TEXT_PTR err_textPtr;      /* return code receiver  */
:                                /* T02 error code text pointer */
{
if ((to2_rc = T02_convertMethodName(env_ptr,
                                   &classNameLength,
                                   className,
                                   &methodNameLength,
                                   methodName)) == T02_ERROR)
{
    to2_rc = T02_getErrorCode(env_ptr);
    err_textPtr = T02_getErrorText(env_ptr, to2_rc);
    printf ("T02_convertMethodName failed, error code - %d\n ", to2_rc);
    printf ("T02 Error Text is %s\n ", err_textPtr);
}
else
    printf("T02_convertMethodName successful\n");
}

```

## Related Information

- “TO2\_getClassDocumentation–Copy Documentation for a Class” on page 1258
- “TO2\_getMethodDocumentation–Copy Documentation for a Method” on page 1291.

## T02\_createDD—Create a Data Definition

This function creates a data definition (DD) and stores it in the data store (DS) dictionary.

### Format

```
#include <c$to2.h>
long T02_createDD (      T02_ENV_PTR    env_ptr,
                        const char      DDname[T02_MAX_DDNAME],
                        const char      dsname[T02_MAX_DSNAME],
                        const T02_OPTION_PTR optionListPtr);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### DDname

The pointer to a character string, which is the name that will be assigned to the created data definition. The character string is assumed to be 32 characters in length. The name should be left-justified and padded with blanks (X'40') on the right if it less than 32 characters.

#### dsname

The pointer to a character string, which is the data store name for which the new DD is being created. The character string must be 8 characters or greater in length, left-justified, and padded with blanks (X'40') on the right.

#### optionListPtr

The pointer to the option list.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

T02\_ERROR\_LOCATOR\_NOT\_UNIQUE

### Programming Considerations

A commit scope will be created for the T02\_createDD request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createDD request, the scope will be rolled back.

### Examples

The following example converts the class name provided into a class index.

```
#include <c$to2.h>                /* Needed for T02 API functions */

long      err_code;
T02_ENV_PTR env_ptr;              /* Pointer to T02 environment */
char      DDname[32];
char      dsname[8];
T02_OPTION_PTR optionListPtr
:
:
if (T02_createDD(env_ptr,
                 DDname,
                 dsname,
```

```
        optionListPtr) == T02_ERROR)  
{  
    err_code = T02_getErrorCode(env_ptr);  
    {  
        printf("T02_createDD failed!\n");  
        process_error(env_ptr);  
    }  
else  
{  
    printf("T02_createDD successful!\n");  
}  
}
```

## Related Information

- “TO2\_changeDD—Change a Data Definition” on page 1230
- “TO2\_deleteDD—Delete a Data Definition” on page 1250
- “TO2\_getDDAttributes—Get the Current DD Attribute Values” on page 1274
- “TO2\_getListDDnames—Retrieve the Defined Data Definition Names” on page 1283.

## T02\_createDS—Create a Data Store

This function creates a data store (DS).

### Format

```
#include <c$to2.h>
long T02_createDS (T02_ENV_PTR env_ptr,
                  const char dsname[T02_MAX_DSNAME]);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### dsname

The pointer to a character string, which is the DS being created. The character string must be 8 characters or less in length, left-justified, and padded with blanks (X'40') on the right.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

A commit scope will be created for the T02\_createDS request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createDS request, the scope will be rolled back.

### Examples

The following example creates a data store with the name indicated.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;       /* Pointer to T02 environment */
char dsname[T02_MAX_DSNAME]="TESTX.DS";
:
:
if (T02_createDS(env_ptr,dsname) == T02_ERROR)
{
    printf("T02_createDS failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_createDS successful!\n");
}
```

### Related Information

- “T02\_deleteDS—Delete a Data Store” on page 1252
- “T02\_getDSAttributes—Get the Attributes of the Data Store” on page 1278
- “T02\_getDSdictPID—Get the Dictionary PID of a Data Store” on page 1212
- “T02\_getDSnameForPID—Determine the DS Name for a PID” on page 1280
- “T02\_getListDSCollections—Retrieve the Data Store System Collections” on page 1285

## **TO2\_createDS**

- “TO2\_getListDSnames–Retrieve the Defined Data Store Names” on page 1287
- “TO2\_migrateCollection–Migrate a Collection” on page 1318
- “TO2\_migrateDS–Migrate a Data Store” on page 1320
- “TO2\_recreateDS–Re-create a Data Store” on page 1330.

See *TPF Application Programming* for more information about commit scope.

## T02\_createDSwithOptions—Create a Data Store with an Option List

This function creates a data store (DS) using a passed option list to modify the normal data store create process.

### Format

```
#include <c$to2.h>
long T02_createDSwithOptions (      T02_ENV_PTR    env_ptr,
                                   const char      dsname[T02_MAX_DSNAME],
                                   const T02_OPTION_PTR optionListPtr);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### dsname

The pointer to a character string, which is the DS being created. The character string must be 8 characters or greater in length, left-justified, and padded with blanks (X'40') on the right.

#### optionListPtr

The pointer to a returned options list from a T02\_createOptionList call.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

A commit scope will be created for the T02\_createDSwithOptions request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_createDSwithOptions request, the scope will be rolled back.

### Examples

The following example creates a data store with the specified name that does not contain an inventory collection and delays the actual deletion of collections.

```
#include <c$to2.h>          /* Needed for T02 API functions      */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* Pointer to T02 environment      */
dsname[] = "TESTDS ";     /* name of DS to create            */
T02_OPTION_PTR optionListPtr;
:

/* invoke T02 to create option list */
optionListPtr=T02_createOptionList(env_ptr,
                                   T02_OPTION_LIST_DS,    /* create options      */
                                   T02_DS_NOINVENTORY,    /* do not create a PID */
                                   /* inventory collection */
                                   T02_DS_DELETE_DELAY,    /* delay collection    */
                                   /* deletions            */
                                   T02_OPTION_LIST_END);    /* end of options      */

if (optionListPtr == T02_ERROR)
    process_error(env_ptr);

if (T02_createDSwithOptions(env_ptr,
```



## TO2\_createDSwithOptions

```
        dsname,  
        optionListPtr) == T02_ERROR)  
{  
    printf("TO2_createDSwithOptions failed!\n");  
    process_error(env_ptr);  
}  
else  
{  
    printf("TO2_createDSwithOptions successful!\n");  
}
```

## Related Information

- “TO2\_changeDS—Change the Attributes of a Data Store” on page 1232
- “TO2\_createDS—Create a Data Store” on page 1240
- “TO2\_createOptionList—Create a TPF Collection Support Option List” on page 990
- “TO2\_getDSAttributes—Get the Attributes of the Data Store” on page 1278
- “TO2\_getDSdictPID—Get the Dictionary PID of a Data Store” on page 1212
- “TO2\_getDSnameForPID—Determine the DS Name for a PID” on page 1280
- “TO2\_getListDSnames—Retrieve the Defined Data Store Names” on page 1287.

## T02\_createPIDinventoryKey—Create PID Inventory Key from PID

This function is used by TPF collection support (TPFCS) utility routines to create a key from an input persistent identifier (PID) that can be used to access the entry of the PID in the PID inventory collection. The created key will be returned in the specified buffer and the length of the key will overlay the **bufferLength** field.

### Format

```
#include <c$to2.h>
long T02_createPIDinventoryKey (T02_PID_PTR  pid_ptr,
                                T02_ENV_PTR  env_ptr,
                                void          *buffer,
                                long          *bufferLength);
```

#### pid\_ptr

The pointer to a field where the temporary PID assigned to the sequence collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### buffer

A pointer to a buffer where the key will be returned.

#### bufferLength

A pointer to a field that contains the length of the buffer. The buffer should be large enough to hold the data to be returned plus 16 additional bytes for a header. This field will be overlaid with the actual length of the key.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_NOT_INIT
```

### Programming Considerations

None.

### Examples

The following example creates the key for a given PID.

```
#include <c$to2.h>          /* Needed for T02 API Functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* Pointer to T02 environment */
T02_PID      collectionPID; /* subject PID */

u_char      buffer[32];
long        bufferLength=sizeof buffer;

T02_ERR_CODE to2_rc=1;     /* return code receiver */
T02_ERR_TEXT_PTR err_textPtr; /* T02 error code text pointer */
```

## T02\_createPIDinventoryKey

```
    :  
{  
if ((to2_rc = T02_createPIDinventoryKey(&collectionPID,  
                                       env_ptr,  
                                       buffer,  
                                       bufferLength)) == T02_ERROR)  
{  
    to2_rc = T02_getErrorCode(env_ptr);  
    err_textPtr = T02_getErrorText(env_ptr, to2_rc);  
    printf ("T02_createPIDinventoryKey failed, error code - %d\n ", to2_rc);  
    printf ("T02 Error Text is %s\n ", err_textPtr);  
}  
else  
    printf("T02_createPIDinventoryKey successful\n");  
}
```

## Related Information

“T02\_getPIDinventoryEntry–Get PID Inventory Entry for PID” on page 1301.

## T02\_defineBrowseNameForPID—Assign a Browse Name to a Collection

This function assigns the given name to the specified collection. Once assigned, the given name can be used on ZBROW functions to access the specified collection. The browse name to persistent identifier (PID) association is maintained in the browser dictionary for the data store (DS) pointed to by the current environment block.

### Format

```
#include <c$to2.h>
long T02_defineBrowseNameForPID (const T02_PID_PTR    pid_ptr,
                                T02_ENV_PTR          env_ptr,
                                const char           name[ ]);
```

#### pid\_ptr

The pointer to the PID assigned to the collection that is to be named.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### name

The pointer to a string, which is the name to be assigned to the collection. The maximum name length is 32 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_DATA_LGH
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

None.

### Examples

The following example assigns the name COLLECTION1 to the input collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;        /* Pointer to the T02 environment */
T02_PID_PTR  collectionPID;  /* named collection's PID */
long         userToken=0;    /* no user token value */
char         applid[]="APPLICATION_NAME";
char         dsname[]="TEST1_DS"; /* data store name */
char         name[] = "COLLECTION1"; /* name to assign */
:
:
/*****
/* The collection must have been created before this function is */
```

## TO2\_defineBrowseNameForPID

```
/* invoked. */
/*****/

if (TO2_createEnv(&env_ptr,
                  &userToken,
                  applID,
                  dsname) == T02_ERROR)
{
    printf("TO2_createEnv failed!\n");
    process_error(env_ptr);
}

if (TO2_defineBrowseNameForPID(&collectionPID,
                               env_ptr,
                               name) == T02_ERROR)
{
    printf("TO2_defineBrowseNameForPID failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_defineBrowseNameForPID successful!\n");
}
```

## Related Information

- “TO2\_deleteBrowseName—Delete a Collection Name from the Browser Dictionary” on page 1248
- “TO2\_getPIDforBrowseName—Retrieve the PID Associated with the Name” on page 1299.

## T02\_deleteBrowseName—Delete a Collection Name from the Browser Dictionary

This function deletes the given name from the browser dictionary for the data store (DS) pointed to by the current environment pointer.

### Format

```
#include <c$to2.h>
long T02_deleteBrowseName (      T02_ENV_PTR  env_ptr,
                               const char    name[ ] );
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### name

The pointer to a string, which is the name to be deleted from the browser dictionary. The maximum name length is 32 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_DATA_LGH
T02_ERROR_METHOD
T02_ERROR_NOT_INIT
T02_ERROR_LOCATOR_NOT_FOUND
```

### Programming Considerations

None.

### Examples

The following example deletes the name COLLECTION1 from the browser dictionary for DS TEST1\_DS.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;        /* Pointer to the T02 environment */
long         userToken=0;    /* no user token value */
char         applid[]="APPLICATION_NAME";
char         dsname[]="TEST1_DS"; /* data store name */
char         name[] = "COLLECTION1"; /* name to assign */
:
:

if (T02_createEnv(&env_ptr,
                 &userToken,
                 applid,
                 dsname)) == T02_ERROR)
{
    printf("T02_createEnv failed!\n");
    process_error(env_ptr);
}
```

## TO2\_deleteBrowseName

```
if (TO2_deleteBrowseName(env_ptr,
                          const name)) == T02_ERROR)
{
    printf("TO2_deleteBrowseName failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_deleteBrowseName was successful!\n");
}
```

### Related Information

- “TO2\_defineBrowseNameForPID—Assign a Browse Name to a Collection” on page 1246
- “TO2\_getPIDforBrowseName—Retrieve the PID Associated with the Name” on page 1299.

## T02\_deleteDD—Delete a Data Definition

This function deletes the data definition (DD).

### Format

```
#include <c$to2.h>
long T02_deleteDD (      T02_ENV_PTR env_ptr,
                        const char    DDname[T02_MAX_DDNAME],
                        const char    dsname[T02_MAX_DSNAME]);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### DDname

The pointer to a character string, which is the name of the DD being deleted. The character string is assumed to be 32 characters in length. The name should be left-justified and padded with blanks (X'40') on the right if it less than 32 characters.

#### dsname

The pointer to a character string, which is the data store (DS) name for the DD being deleted. The character string must be 8 characters or greater in length, left-justified, and padded with blanks (X'40') on the right.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_DD_NOT_FOUND
T02_ERROR_LOCATOR_NOT_FOUND
```

### Programming Considerations

- Data definition names that begin with DEFAULT\_ cannot be deleted.
- A commit scope will be created for the T02\_deleteDD request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_deleteDD request, the scope will be rolled back.

### Examples

The following example deletes a data store with the name indicated.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;      /* Pointer to T02 environment */
char      DDname[32];
char      dsname[8];
:
:
if (T02_deleteDD(env_ptr,
                 DDname,
                 dsname) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
}
```



```
{
    printf("TO2_deleteDD failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_deleteDD successful!\n");
}
}
```

## Related Information

- “TO2\_changeDD—Change a Data Definition” on page 1230
- “TO2\_createDD—Create a Data Definition” on page 1238
- “TO2\_getDDAttributes—Get the Current DD Attribute Values” on page 1274
- “TO2\_getListDDnames—Retrieve the Defined Data Definition Names” on page 1283.

## T02\_deleteDS—Delete a Data Store

This function deletes a data store (DS).

### Format

```
#include <c$to2.h>
long T02_deleteDS (T02_ENV_PTR env_ptr,
                  const char dsname[T02_MAX_DSNAME]);
```

#### env\_ptr

A pointer to the environment as returned by the T02\_createEnv function.

#### dsname

A pointer to a character string, which represents the data store being deleted. The character string must be 8 characters or less in length, left-justified, and padded with blanks (X'40') on the right.

### Normal Return

A positive value.

### Error Return

A value of zero. Use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

## Programming Considerations

- This function deletes the data store entry from the TPF system dictionary and deletes all system collections in that data store. All resources used by these collections are returned to the system and none of these collections can be reclaimed.
- You do **not** have to call this function from the subsystem that created the data store.
- Deleting a data store does not delete user collections. You must delete all user collections before calling this function or the pool records used by these collections will be lost until the next recoup is run.
- You cannot delete the TPFDB data store.
- A commit scope will be created for the T02\_deleteDS request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_deleteDS request, the scope will be rolled back.

## Examples

The following example deletes data store TESTX.DS.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;       /* Pointer to T02 environment */
char dsname[T02_MAX_DSNAME]="TESTX.DS";
:
:
if (T02_deleteDS(env_ptr, dsname) == T02_ERROR)
{
    printf("T02_deleteDS failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_deleteDS successful!\n");
}
```

## Related Information

- “TO2\_createDS—Create a Data Store” on page 1240
- “TO2\_getDSAttributes—Get the Attributes of the Data Store” on page 1278
- “TO2\_getDSdictPID—Get the Dictionary PID of a Data Store” on page 1212
- “TO2\_getDSnameForPID—Determine the DS Name for a PID” on page 1280
- “TO2\_getListDSCollections—Retrieve the Data Store System Collections” on page 1285
- “TO2\_getListDSnames—Retrieve the Defined Data Store Names” on page 1287
- “TO2\_migrateDS—Migrate a Data Store” on page 1320
- “TO2\_recreateDS—Re-create a Data Store” on page 1330.

See *TPF Application Programming* for more information about commit scope. See *TPF Database Reference* for more information about data stores.

## T02\_getBrowseDictPID–Get the Browser Dictionary PID

This function finds the persistent identifier (PID) of the TPFCS browser dictionary.

### Format

```
#include <c$to2.h>
long T02_getBrowseDictPID (T02_PID_PTR  pid_ptr,
                          T02_ENV_PTR  env_ptr);
```

#### pid\_ptr

The pointer to where to return the PID assigned to the TPFCS browser dictionary.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

None.

### Examples

The following example retrieves the PID for the browser dictionary indicated.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;      /* Pointer to T02 environment */
T02_PID_PTR collect;
:
:
if (T02_getBrowseDictPID(&collect,
                        env_ptr) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    {
        printf("T02_getBrowseDictPID failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_getBrowseDictPID successful!\n");
}
}
```

### Related Information

- “T02\_atBrowseKey–Retrieve the Element Using the Specified Key from the Browser Dictionary” on page 1224
- “T02\_atBrowseKeyPut–Replace the Element Using the Specified Key from the Browser Dictionary” on page 1226
- “T02\_atBrowseNewKeyPut–Store the Element Using the Specified Key in the Browser Dictionary” on page 1228

- "TO2\_removeBrowseKey–Remove the Element Using the Specified Key" on page 1332 .

## T02\_getClassAttributes—Get Attributes of a Class

This function returns a sequence collection of the EBCDIC names of the class attributes and the values for the class.

### Format

```
#include <c$to2.h>
long T02_getClassAttributes (T02_PID_PTR    pid_ptr,
                           T02_ENV_PTR    env_ptr,
                           const long    *class_namlength,
                           const char    class_name[T02_MAX_CLASS_NAME]);
```

#### pid\_ptr

The pointer to where to return the temporary persistent identifier (PID) assigned to the sequence collection of attributes.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### class\_namlength

The pointer to a long integer, which contains the length of the input class name.

#### class\_name

The character string class name that will be interrogated.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_NO_ROOM
T02_ERROR_INVALID_DATA_LGH
```

### Programming Considerations

None.

### Examples

The following example retrieves the PID for the class name indicated.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

long    err_code;
T02_ENV_PTR env_ptr;      /* Pointer to T02 environment */
T02_PID_PTR collect;
char    *class_name;
long    classname_length;
:
:
if (T02_getClassAttributes(&collect,
                        env_ptr,
                        &classname_length,
                        class_name) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    {
        printf("T02_getClassAttributes failed!\n");
    }
}
```

```
        process_error(env_ptr);
    }
    else
    {
        printf("TO2_getClassAttributes successful!\n");
    }
}
```

## Related Information

- “TO2\_getClassInfo—Get Information for a Class” on page 1260
- “TO2\_getClassNames—Convert EBCDIC Class Name to Index” on page 1262
- “TO2\_getClassTree—Get an Inheritance Tree for a Class” on page 1264.

## T02\_getClassDocumentation—Copy Documentation for a Class

This function causes TPF collection support (TPFCS) to locate the specific class definition in storage and return the embedded class documentation in a temporary sequence collection. If there is no class documentation, the returned sequence collection will be empty.

### Format

```
#include <c$to2.h>
long T02_getClassDocumentation (      T02_PID_PTR  pid_ptr,
                                     T02_ENV_PTR  env_ptr,
                                     const long    *class_namlength,
                                     const char     class_name[T02_MAX_CLASS_NAME]);
```

#### pid\_ptr

The pointer to a field where the temporary persistent identifier (PID) assigned to the sequence collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### class\_namlength

A pointer to a field that contains the length of the class name.

#### class\_name

A pointer to a field that contains the class name.

### Normal Return

The normal return is a positive value. The returned sequence collection will be empty if no class documentation existed for the class.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CLASS_NOTFOUND
TO2_ERROR_ENV
TO2_ERROR_NOT_INIT
```

## Programming Considerations

The following shows how the elements in the return collection will be structured:

```
struct T02_class_doc_line
{
    char    text[80];    /* documentation text */
};
```

### Examples

The following example copies the class documentation for class OBJECT.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 environment */
T02_PID        collectionPID;    /* return area for PID */

char           class_name[]="OBJECT"; /* class name */
```



## T02\_getClassDocumentation

```
long          class_namelength=(sizeof "OBJECT")-1; /* length          */
/* class name          */
T02_ERR_CODE  to2_rc=1; /* return code receiver          */
T02_ERR_TEXT_PTR err_textPtr; /* T02 error code text pointer          */
:
{
if ((to2_rc = T02_getClassDocumentation(&collectionPID,
                                       env_ptr,
                                       &class_namelength,
                                       class_name)) == T02_ERROR)

{
    to2_rc = T02_getErrorCode(env_ptr);
    err_textPtr = T02_getErrorText(env_ptr, to2_rc);
    printf ("T02_getClassDocumentation failed, error code - %d\n ", to2_ rc);
    printf ("T02 Error Text is %s\n ", err_textPtr);
}
else
    printf("T02_getClassDocumentation successful\n");
}
```

## Related Information

"T02\_getMethodDocumentation–Copy Documentation for a Method" on page 1291.

## T02\_getClassInfo—Get Information for a Class

This function returns a sequence collection of the information about the specified class.

### Format

```
#include <c$to2.h>
long T02_getClassInfo (      T02_PID_PTR   pid_ptr,
                             T02_ENV_PTR   env_ptr,
                             const long    *class_nameLength,
                             const char    class_name[T02_MAX_CLASS_NAME]);
```

#### **pid\_ptr**

The pointer to where to store the temporary persistent identifier (PID) assigned to the sequence collection of class information.

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **class\_nameLength**

The pointer to the length of the input class name.

#### **class\_name**

The character string class name that will be interrogated.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

T02\_ERROR\_NO\_ROOM

### Programming Considerations

None.

### Examples

The following example retrieves the PID for the class name indicated.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;        /* Pointer to T02 environment */
T02_PID_PTR collect;
char      *class_name;
long      classname_length;
:
:
if (T02_getClassInfo(&collect,
                    env_ptr,
                    &classname_length,
                    class_name) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    {
        printf("T02_getClassInfo failed!\n");
        process_error(env_ptr);
    }
}
```

```
}  
else  
{  
    printf("T02_getClassInfo successful!\n");  
}  
}
```

## **Related Information**

- “TO2\_getClassAttributes—Get Attributes of a Class” on page 1256
- “TO2\_getClassNames—Convert EBCDIC Class Name to Index” on page 1262
- “TO2\_getClassTree—Get an Inheritance Tree for a Class” on page 1264.

## T02\_getClassNames—Convert EBCDIC Class Name to Index

This function returns a sequence collection of the EBCDIC names of all defined classes to the caller.

### Format

```
#include <c$to2.h>
long T02_getClassNames (T02_PID_PTR   pid_ptr,
                       T02_ENV_PTR   env_ptr);
```

#### pid\_ptr

The pointer to where to store the temporary persistent identifier (PID) assigned to the sequence collection of class names.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

None.

### Examples

The following example retrieves a sequence collection of all defined classes.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;        /* Pointer to T02 environment */
T02_PID_PTR collect;
:
:
if (T02_getClassNames(&collect,
                    env_ptr) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    {
        printf("T02_getClassNames failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_getClassNames successful!\n");
}
}
```

### Related Information

- “T02\_getClassAttributes—Get Attributes of a Class” on page 1256
- “T02\_getClassInfo—Get Information for a Class” on page 1260
- “T02\_getClassTree—Get an Inheritance Tree for a Class” on page 1264
- “T02\_getCollectionAttributes—Get the Attributes of the Collection” on page 1266

- "TO2\_getMethodNames–Get Method Names for a Class" on page 1293.

## T02\_getClassTree—Get an Inheritance Tree for a Class

This function returns a sequence collection of the EBCDIC class names from which the specified class inherits.

### Format

```
#include <c$to2.h>
long T02_getClassTree (      T02_PID_PTR  pid_ptr,
                             T02_ENV_PTR  env_ptr,
                             const long    *class_namlength,
                             const char    class_name[T02_MAX_CLASS_NAME]);
```

#### pid\_ptr

The pointer to where to store the temporary persistent identifier (PID) assigned to the sequence collection of class names.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### class\_namlength

The pointer to a long integer, which contains the length of the input class name.

#### class\_name

The character string class name that will be converted.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

None.

### Examples

The following example retrieves a sequence collection of inherited classes.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;      /* Pointer to T02 environment */
T02_PID_PTR collect;
char      *class_name;
long      class_namlength;
:
:
if (T02_getClassTree(&collect,
                    env_ptr,
                    &class_namlength,
                    class_name) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    {
        printf("T02_getClassTree failed!\n");
        process_error(env_ptr);
    }
}
else
```

```
{  
    printf("TO2_getClassTree successful!\n");  
}
```

## **Related Information**

"TO2\_getClassNames—Convert EBCDIC Class Name to Index" on page 1262.

## T02\_getCollectionAttributes—Get the Attributes of the Collection

This function returns a sequence collection of the EBCDIC names of the attributes of the collection and their current values.

### Format

```
#include <c$to2.h>
long T02_getCollectionAttributes (const T02_PID_PTR pid_ptr,
                                   T02_ENV_PTR env_ptr,
                                   T02_PID_PTR retn_pid_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) of the collection whose attributes will be returned in a sequence collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### retn\_pid\_ptr

The pointer to where to return the temporary PID assigned to the sequence collection of attributes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

None.

### Examples

The following example retrieves the sequence collection of method names for the data store (DS) indicated.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;        /* Pointer to T02 environment */
T02_PID_PTR pid_ptr;
T02_PID_PTR retn_pid_ptr;
:
:
if (T02_getCollectionAttributes(&collect,
                               env_ptr,
                               &retn_pid_ptr) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    {
        printf("T02_getCollectionAttributes failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_getCollectionAttributes successful!\n");
}
}
```



## **Related Information**

"TO2\_getClassNames—Convert EBCDIC Class Name to Index" on page 1262.

## T02\_getCollectionName—Get the Class Name of a Collection

This function returns the class name of the specified collection.

### Format

```
#include <c$to2.h>
T02_BUF_PTR T02_getCollectionName (const T02_PID_PTR  pid_ptr,
                                     T02_ENV_PTR   env_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) of the collection whose class name is to be retrieved.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a pointer (T02\_BUF\_PTR) to a structure (buffer) of type T02\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

- The returned collection name is not guaranteed to be NULL delimited. You should use the **dataL** field in the buffer header to determine the length of the returned character string.
- The caller must free the returned buffer once the caller has stopped using the buffer. Enter a free function to free the buffer.

### Examples

The following example retrieves the class name for the collection indicated.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* Pointer to T02 environment */
T02_PID_PTR  collect;
T02_BUF_PTR  bufferPtr;
char         *collectionName;
long         nameLength;
:
:
if ((bufferPtr = T02_getCollectionName(&collect,
                                     env_ptr)) == T02_ERROR)
{
    printf("T02_getCollectionName failed!\n");
    process_error(env_ptr);
}
else
{
    collectionName = bufferPtr->data;
    nameLength = bufferPtr->dataL;
    printf("T02_getCollectionName successful!\n");
}
:
free(buffer_ptr);
```

## Related Information

"TO2\_getClassNames—Convert EBCDIC Class Name to Index" on page 1262.

## T02\_getCollectionParts—Get the Part Names of the Collection

This function returns a temporary sequence collection containing the class names of the collection parts that were used to compose the specified collection.

### Format

```
#include <c$to2.h>
long T02_getCollectionParts (T02_PID_PTR pid_ptr,
                           T02_ENV_PTR env_ptr,
                           T02_PID_PTR retn_pid_ptr);
```

#### pid\_ptr

The pointer to a field that contains the persistent identifier (PID) of the collection that will be interrogated.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### retn\_pid\_ptr

The pointer to a field where the PID of the temporary collection will be returned.

### Normal Return

The normal return is a positive value and the specified T02\_PID\_PTR field contains the PID of the temporary sequence collection.

The sequence collection will contain an element in the following format for each part:

```
struct T02_partList_element {      /*
    long    partNumber;           /* part number of this part  */
    u_char   partNameLgh;         /* length of the part's name  */
    char     partName[ ];         /* object part's class name   */
};
```

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_NOT_INIT
T02_ERROR_PID
```

### Programming Considerations

None.

### Examples

The following example copies the part class names for a collection.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                 /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;           /* Pointer to T02 environment */
T02_PID        objectPID;         /* target object's PID */
T02_PID        collectionPID;     /* return area for PID */

T02_ERR_CODE    to2_rc=1;         /* return code receiver */
T02_ERR_TEXT_PTR err_textPtr;     /* T02 error code text pointer */
```

```
    :
    {
    if ((to2_rc = T02_getCollectionParts(&objectPID,
                                       env_ptr,
                                       collectionPidPtr)) == T02_ERROR)
    {
        to2_rc = T02_getErrorCode(env_ptr);
        err_textPtr = T02_getErrorText(env_ptr, to2_rc);
        printf ("T02_getBLOB failed, error code - %d\n ", to2_rc);
        printf ("T02 Error Text is %s\n ", err_textPtr);
    }
    else
        printf("T02 get object parts successful\n");
    }
```

## Related Information

"T02\_getClassNames—Convert EBCDIC Class Name to Index" on page 1262.

## T02\_getCreateTime—Copy Created Time Stamp of the Collection

Whenever TPF collection support (TPFCS) creates a collection, it will time stamp the collection with the current time. This function copies the created time value from the specified collection and returns it in the field provided.

### Format

```
#include <c$to2.h>
long T02_getCreateTime (const T02_PID_PTR pid_ptr,
                           T02_ENV_PTR env_ptr,
                           void *timeRetPtr);
```

#### pid\_ptr

A pointer to the persistent identifier (PID) of the collection whose created time is to be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### timeRetPtr

A pointer to an 8-byte field where the created time of the collection will be returned. The returned time is in System/390 time-of-day (TOD) clock format.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
TO2_ERROR_DELETED_PID
TO2_ERROR_NOT_INIT
```

### Programming Considerations

None.

### Examples

The following example copies the created time value for the provided collection.

```
#include <c$to2.h> /* Needed for T02 API functions */
long getCreationTime(const T02_PID_PTR pid_ptr,
                       T02_ENV_PTR env_ptr,
                       double * timeRetPtr)

/*
/* copy the create time value from the input collection
/* and return it to the caller.
/* returns 0 if successful otherwise -1 if an error occurs.
/*
{
:
:
if (T02_getCreateTime(pid_ptr,
                     env_ptr,
                     timeRetPtr) == T02_ERROR)
{
    process_error(env_ptr);
}
```

```
    return -1;  
}  
    return 0;  
}
```

## **Related Information**

None.

## T02\_getDDAttributes—Get the Current DD Attribute Values

This function returns a sequence collection of the EBCDIC names of the specified attributes of the data definition (DD) and their current values.

### Format

```
#include <c$to2.h>
long T02_getDDAttributes (      T02_PID_PTR  pid_ptr,
                                T02_ENV_PTR  env_ptr,
                                const char    DDname[T02_MAX_DDNAME],
                                const char    dsname[T02_MAX_DSNAME]);
```

#### pid\_ptr

The pointer to where to return the temporary persistent identifier (PID) assigned to the sequence collection of attributes.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### DDname

The pointer to a character string, which is the name of the DD being interrogated. The character string is assumed to be 32 characters in length. The name should be left-justified and padded with blanks (X'40') on the right if it less than 32 characters.

#### dsname

The pointer to a character string, which is the data store (DS) name for the DD whose attributes are being retrieved. The character string must be 8 characters or greater in length, left-justified, and padded with blanks (X'40') on the right.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

None.

### Examples

The following example retrieves a sequence collection of attributes for the DD name indicated.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */
long      err_code;
T02_ENV_PTR env_ptr;      /* Pointer to T02 environment */
T02_PID_PTR collect;
char      DDname[32];
char      dsname[8];
:
:
if (T02_getDDAttributes(&collect,
                       env_ptr,
                       DDname,
                       dsname) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
}
```



```
{
    printf("TO2_getDDAttributes failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_getDDAttributes successful!\n");
}
}
```

## Related Information

- “TO2\_changeDD—Change a Data Definition” on page 1230
- “TO2\_createDD—Create a Data Definition” on page 1238
- “TO2\_deleteDD—Delete a Data Definition” on page 1250
- “TO2\_getListDDnames—Retrieve the Defined Data Definition Names” on page 1283.

## T02\_getDirectoryForRRN—Get the Current Directory for the Specified RRN

This function is used to locate and return the directory for the specified relative record number (RRN) for the target persistent identifier (PID).

### Format

```
#include <c$to2.h>
long T02_getDirectoryForRRN (T02_PID_PTR pid_ptr,
                             T02_ENV_PTR env_ptr,
                             long rrn,
                             void *directoryPtr);
```

#### pid\_ptr

The pointer to the PID of the collection that will be interrogated.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### rrn

The relative record number value.

#### directoryPtr

The pointer to the 24-byte output area for the directory.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_PID
T02_ERROR_UNASSIGNED_RRN
```

### Programming Considerations

None.

### Examples

This example shows how the T02\_getDirectoryForRRN function retrieves the directory for an RRN that is passed as input.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */
T02_ENV_PTR env_ptr;        /* environment pointer */
T02_PID_PTR collect;        /* PID of the target collection */
long rrn;                   /* variable to hold input RRN */
char directory[25];         /* field for TPFCS to return */
                             /* directory contents. */

/* Invoke TPFCS to obtain the directory entry of the relative */
/* record number specified. */

if (T02_getDirectoryForRRN(&collect,
                          env_ptr,
                          rrn,
```

## TO2\_getDirectoryForRRN

```
        directory) == T02_ERROR)  
{  
    printf("T02_getDirectoryForRRN failed!\n");  
    process_error(env_ptr);  
}  
else  
{  
    printf("T02_getDirectoryForRRN successful!\n");  
}
```

### Related Information

None.

## TO2\_getDSAttributes—Get the Attributes of the Data Store

This function returns a temporary sequence collection containing the names and values of the defined attributes for the specified data store (DS).

### Format

```
#include <c$to2.h>
long TO2_getDSAttributes (      T02_PID_PTR  pid_ptr,
                               T02_ENV_PTR  env_ptr,
                               const char    dsname[T02_MAX_DSNAME]);
```

#### pid\_ptr

A pointer to a field where the persistent identifier (PID) of the temporary collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### dsname

A pointer to the name of the DS with attributes that will be returned. The character string must be 8 characters or greater in length, left-justified, and padded with blanks (X'40') on the right.

### Normal Return

The normal return is a positive value and the specified TO2\_PID\_PTR field will contain the PID of the temporary sequence collection.

The sequence collection will contain an element in the following format for each defined attribute:

```
enum T02_ATTRIBUTE_FORMAT
{
    T02_ATTRIBUTE_CHAR = 'C',    /* format as character string */
    T02_ATTRIBUTE_DEC  = 'D',    /* format as decimal number   */
    T02_ATTRIBUTE_HEX  = 'H' }; /* format as hexadecimal value */

struct T02_attribute_element
{
    /* */
    enum T02_ATTRIBUTE_FORMAT format_type; /* how to format */
    /* attribute value. */
    u_char attr_val_lgh; /* length of attribute field */
    u_char attr_nam_lgh; /* length of attribute name */
    char attr_name[64]; /* attribute name
                        /* attribute name = C'*****' is
                        /* value continuation line for
                        /* previously named attribute.
    u_char attr_value[32]; /* attribute value
};
```

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_DBID
TO2_ERROR_NOT_INIT
```

## Programming Considerations

None.

## Examples

The following example copies the attributes for the TPFDB data store.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */
T02_ENV_PTR    env_ptr;          /* Pointer to T02 environment */
T02_PID        collectionPID;    /* return area for PID */

char           DSname[9]        PFDB  "; /* data store name blank padded */
T02_ERR_CODE   to2_rc=1;        /* return code receiver */
T02_ERR_TEXT_PTR err_textPtr;    /* T02 error code text pointer */
:
{
if ((to2_rc = T02_getDSAttributes(&collectionPID,
                                env_ptr,
                                DSname)) == T02_ERROR)
{
    to2_rc = T02_getErrorCode(env_ptr);
    err_textPtr = T02_getErrorText(env_ptr, to2_rc);
    printf ("T02_getDSAttributes failed, error code - %d\n ", to2_rc);
    printf ("T02 Error Text is %s\n ", err_textPtr);
}
else
    printf("T02 get DS attributes successful\n");
}
```

## Related Information

- “TO2\_createDS—Create a Data Store” on page 1240
- “TO2\_getListDSnames—Retrieve the Defined Data Store Names” on page 1287.

### TO2\_getDSnameForPID—Determine the DS Name for a PID

This function determines the data store (DS) name for a persistent identifier (PID).

#### Format

```
#include <c$to2.h>
long TO2_getDSnameForPID(const T02_PID_PTR pid_ptr,
                          T02_ENV_PTR env_ptr,
                          char dsname[T02_MAX_DSNAME] );
```

##### **pid\_ptr**

The pointer to the PID whose DS name will be determined.

##### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

##### **dsname**

The return field for the DS name. The character string must be 8 characters or greater in length.

#### Normal Return

The normal return is a positive value.

#### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

#### Programming Considerations

None.

#### Examples

The following example retrieves the DS name for the PID indicated.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */
long err_code;
T02_ENV_PTR env_ptr;       /* Pointer to T02 environment */
T02_PID collect;
char dsname[8];
:
:
if (TO2_getDSnameForPID(&collect,
                      env_ptr,
                      dsname) == T02_ERROR)
{
    printf("TO2_getDSnameForPID failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_getDSnameForPID successful!\n");
}
```

#### Related Information

- “TO2\_createDS—Create a Data Store” on page 1240
- “TO2\_getDSAttributes—Get the Attributes of the Data Store” on page 1278
- “TO2\_getDSdictPID—Get the Dictionary PID of a Data Store” on page 1212
- “TO2\_getListDSnames—Retrieve the Defined Data Store Names” on page 1287.

## TO2\_getKeyPathAttributes—Retrieve the Attributes of Key Paths

This function allows the TPF collection support (TPFCS) browser to retrieve the attributes of a specific alternate key path or all alternate key paths for a specific collection.

### Format

```
#include <c$to2.h>
long TO2_getKeyPathAttributes (const T02_PID_PTR pid_ptr,
                                T02_ENV_PTR env_ptr,
                                T02_PID_PTR returnPid_ptr,
                                const char *name);
```

#### **pid\_ptr**

The pointer to the persistent identifier (PID) assigned to the target collection.

#### **env\_ptr**

The pointer to the environment as returned by the TO2\_createEnv function.

#### **returnPid\_ptr**

The pointer to an area to hold the temporary PID of the returned sequence collection.

#### **name**

The pointer to a string that is the name of the alternate key path whose attributes will be retrieved or NULL if the attributes of all key paths will be retrieved.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- This function does not retrieve the attributes of the primary key path. The TO2\_ERROR\_PARAMETER error code will be returned if the TO2\_PRIME\_KEYPATH name is passed to this function.
- This function will return TO2\_ERROR\_METHOD if the target collection does not support key paths.
- This function will return an empty collection if the target collection does not have any alternate key paths defined.

### Examples

The following example retrieves the attributes of the specified alternate key path.

## TO2\_getDSnameForPID

```
#include <c$to2.h>           /* T02 API function prototypes */
#include <stdio.h>           /* APIs for standard I/O functions */
T02_ENV_PTR    env_ptr;     /* Pointer to T02 Environment */
T02_PID        keyset;
T02_PID        returnPid_ptr;
char           name="fieldA"; /* name of key path to remove */
:
:
if (T02_getKeyPathAttributes(&keyset,
                             env_ptr,
                             &returnPid_ptr,
                             name) == T02_ERROR)
{
    printf("T02_getKeyPathAttributes failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_getKeyPathAttributes was successful!\n");
}
```

## Related Information

- “TO2\_addKeyPath—Add a Key Path to a Collection” on page 890
- “TO2\_removeKeyPath—Remove a Key Path from a Collection” on page 1082
- “TO2\_setKeyPath—Set a Cursor to Use a Specific Key Path” on page 1180.



## T02\_getListDDnames—Retrieve the Defined Data Definition Names

This function returns a sequence collection of defined data definition (DD) names that are defined for the data store (DS) associated with the passed TPF collection support(TPFCS) environment block.

### Format

```
#include <c$to2.h>
long T02_getListDDnames (T02_PID_PTR pid_ptr,
                        T02_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to the field to hold the returned temporary persistent identifier (PID) assigned to the created temporary sequence collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function. DD names will be returned for the data store that is associated with the environment pointer **env\_ptr** passed as a parameter.

### Normal Return

The sequence collection will contain one element per defined data definition. The format of the data area of the element is as follows:

```
struct ddElement
{
    char    dataStoreName[32];
}
```

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

None.

### Examples

The following example retrieves the sequence collection of DD names for the PID indicated.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */
long    err_code;
T02_ENV_PTR env_ptr;        /* Pointer to T02 environment */
T02_PID_PTR collect;
:
if (T02_getListDDnames(&collect,
                      env_ptr) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    {
        printf("T02_getListDDnames failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_getListDDnames successful!\n");
}
}
```

## TO2\_getListDDnames

### Related Information

- “TO2\_changeDD—Change a Data Definition” on page 1230
- “TO2\_createDD—Create a Data Definition” on page 1238
- “TO2\_deleteDD—Delete a Data Definition” on page 1250
- “TO2\_getDDAttributes—Get the Current DD Attribute Values” on page 1274
- “TO2\_getListDSnames—Retrieve the Defined Data Store Names” on page 1287.

## T02\_getListDSCollections—Retrieve the Data Store System Collections

This function returns a temporary sequence collection that contains a list of the persistent identifiers (PIDs) for the system collections in a specified data store.

### Format

```
#include <c$to2.h>
long T02_getListDSCollections (T02_PID_PTR    pid_ptr,
                              T02_ENV_PTR    env_ptr,
                              const char     dsname[T02_MAX_DSNAME]);
```

#### pid\_ptr

A pointer to the field that holds the returned temporary PID assigned to the temporary sequence collection created in response to this request.

#### env\_ptr

A pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

A positive value. The sequence collection referred to by **pid\_ptr** contains one element per system collection. The format of the data area of the element is as follows:

```
struct T02_dsCollection_element
{
    T02_PID collectionPID[T02_MAX_PID_SIZE];
};
```

### Error Return

A value of zero. Use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

Enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection even though the temporary collection is automatically deleted when the entry control block (ECB) exits. This action ensures that system resources used by that collection are cleanly released back to the system for reuse.

### Examples

The following example retrieves the sequence collection of data store system collections for data store CUSTOMER.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_PID    pid;            /* Placeholder for temporary collect */
T02_ENV_PTR env_ptr;       /* Pointer to T02 environment */
char       dsname[T02_MAX_DSNAME]="CUSTOMER";
:
:
if (T02_getListDSCollections(&pid, env_ptr, dsname) ==T02_ERROR)
{
    printf("T02_getListDSCollections failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_getListDSCollections successful!\n");
}
```

## TO2\_getListDSCollections

### Related Information

- “TO2\_createDS—Create a Data Store” on page 1240
- “TO2\_deleteCollection—Delete a Collection” on page 1036
- “TO2\_deleteDS—Delete a Data Store” on page 1252
- “TO2\_getDSAttributes—Get the Attributes of the Data Store” on page 1278
- “TO2\_getDSdictPID—Get the Dictionary PID of a Data Store” on page 1212
- “TO2\_getDSnameForPID—Determine the DS Name for a PID” on page 1280
- “TO2\_getListDSNames—Retrieve the Defined Data Store Names” on page 1287
- “TO2\_migrateDS—Migrate a Data Store” on page 1320
- “TO2\_recreateDS—Re-create a Data Store” on page 1330.

## T02\_getListDSnames—Retrieve the Defined Data Store Names

This function returns a sequence collection of defined data store (DS) names and other associated information about the data store.

### Format

```
#include <c$to2.h>
long T02_getListDSnames (T02_PID_PTR pid_ptr,
                        T02_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to the field to hold the returned temporary persistent identifier (PID) assigned to the created temporary sequence collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The sequence collection contains one element per defined data store. The format of the data area of the element is as follows:

```
struct dsElement
{
    char    dataStoreName[8];
    char    SSname[4];
    u_char  inventory_pid[32];
}
```

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

None.

### Examples

The following example retrieves the sequence collection of data store names for the PID indicated.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

long    err_code;
T02_ENV_PTR env_ptr;      /* Pointer to T02 environment */
T02_PID_PTR collect;
:
if (T02_getListDSnames(&collect,
                      env_ptr) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);

    printf("T02_getListDSnames failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_getListDSnames successful!\n");
}
```

## TO2\_getListDSnames

### Related Information

- “TO2\_createDS—Create a Data Store” on page 1240
- “TO2\_deleteDS—Delete a Data Store” on page 1252
- “TO2\_getDSAttributes—Get the Attributes of the Data Store” on page 1278
- “TO2\_getDSdictPID—Get the Dictionary PID of a Data Store” on page 1212
- “TO2\_getDSnameForPID—Determine the DS Name for a PID” on page 1280
- “TO2\_getListDDnames—Retrieve the Defined Data Definition Names” on page 1283
- “TO2\_getListDScollections—Retrieve the Data Store System Collections” on page 1285.

## T02\_getListUsers—Get a List of Defined Users

This function returns a temporary sequence collection containing a list of all users defined to TPF collection support (TPFCS).

### Format

```
#include <c$to2.h>
long T02_getListUsers (T02_PID_PTR pid_ptr,
                      T02_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to a field where the persistent identifier (PID) of the temporary collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value and the specified T02\_PID\_PTR field contains the PID of the temporary sequence collection.

The sequence collection will contain an element in the following format for each user:

```
struct T02_listUsers_element
{
    u_long    userToken;        /* assigned user token
    char      userID[32];       /* user ID
    char      applID[32];       /* application ID
    T02_PID   userPID;          /* user object PID
};
```

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_NOT_INIT
```

### Programming Considerations

When the caller is done with the returned temporary collection, delete the collection by using the T02\_deleteCollection function.

### Examples

The following example copies the class names for a collection.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                 /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;           /* Pointer to T02 environment */
T02_PID        userListPID;       /* return area for PID */

T02_ERR_CODE   to2_rc=1;          /* return code receiver */
T02_ERR_TEXT_PTR err_textPtr;     /* T02 error code text pointer */
:
{
if ((to2_rc = T02_getListUsers(&userListPidPtr,
```

## T02\_getListUsers

```
env_ptr)) == T02_ERROR)
{
    to2_rc = T02_getErrorCode(env_ptr);
    err_textPtr = T02_getErrorText(env_ptr, to2_rc);
    printf ("T02_getListUsers failed, error code - %d\n ", to2_rc);
    printf ("persistent collections Error Text is %s\n ", err_textPtr);
}
else
    printf("persistent collections get user list was successful\n");
}
```

## Related Information

None.



## TO2\_getMethodDocumentation–Copy Documentation for a Method

This function causes TPF collection support (TPFCS) to locate the specific method definition in storage and return the embedded method documentation in a temporary sequence collection. If there is no method documentation, the returned sequence collection will be empty.

### Format

```
#include <c$to2.h>
long TO2_getMethodDocumentation (      T02_PID_PTR pid_ptr,
                                      T02_ENV_PTR env_ptr,
                                      const long    *class_namelen,
                                      const char     class_name[T02_MAX_CLASS_NAME],
                                      const long    *method_namelen,
                                      const char     method_name[T02_MAX_METHOD_NAME]);
```

#### pid\_ptr

The pointer to a field where the temporary persistent identifier (PID) assigned to the sequence collection will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### class\_namelen

A pointer to a field that contains the length of the class name.

#### class\_name

A pointer to a field that contains the class name.

#### method\_namelen

A pointer to a field that contains the length of the method name.

#### method\_name

A pointer to a field that contains the method name.

### Normal Return

The normal return is a positive value. The returned sequence collection will be empty if no method documentation existed for the method.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_CLASS_NOTFOUND
TO2_ERROR_ENV
TO2_ERROR_METHOD_NOTFOUND
TO2_ERROR_NOT_INIT
```

### Programming Considerations

The elements in the return collection will be of the following structure:

```
struct T02_method_doc_line
{
    char    text[80];          /* documentation text */
};
```

## T02\_getMethodDocumentation

### Examples

The following example copies the method documentation for class OBJECT method new.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 environment */
T02_PID        collectionPID;    /* return area for PID */

char           class_name[]="OBJECT"; /* class name */
long           class_namelen=(sizeof "OBJECT")-1; /* length of class name */
char           method_name[]="new"; /* method name */
long           method_namelen=(sizeof "new")-1; /* length of method name */
T02_ERR_CODE   to2_rc=1;         /* return code receiver */
T02_ERR_TEXT_PTR err_textPtr;    /* T02 error code text pointer */
:
{
if ((to2_rc = T02_getMethodDocumentation(&collectionPID,
                                       env_ptr,
                                       &class_namelen,
                                       class_name,
                                       &method_namelen,
                                       method_name)) == T02_ERROR)
{
to2_rc = T02_getErrorCode(env_ptr);
err_textPtr = T02_getErrorText(env_ptr, to2_rc);
printf ("T02_getMethodDocumentation failed, error code - %d\n ", to2 _rc);
printf ("T02 Error Text is %s\n ", err_textPtr);
}
else
printf("T02_getMethodDocumentation successful\n");
}
```

### Related Information

"T02\_getClassDocumentation—Copy Documentation for a Class" on page 1258.

## T02\_getMethodNames—Get Method Names for a Class

This function returns a sequence collection of the EBCDIC method names of the specified class to the caller.

### Format

```
#include <c$to2.h>
long T02_getMethodNames (      T02_PID_PTR  pid_ptr,
                               T02_ENV_PTR  env_ptr,
                               const long    *class_namelenh,
                               const char    class_name[T02_MAX_CLASS_NAME]);
```

#### **pid\_ptr**

The pointer to where to store the temporary persistent identifier (PID) assigned to the sequence collection of method names.

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **class\_namelenh**

The pointer to a long integer, which contains the length of the input class name.

#### **class\_name**

The character string class name that will be converted.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

None.

### Examples

The following example retrieves the sequence collection of method names for the data store (DS) indicated.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;        /* Pointer to T02 environment */
T02_PID_PTR collect;
char      *class_name;
long      class_namelenh;
:
:
if (T02_getMethodNames(&collect,
                      env_ptr,
                      &class_namelenh,
                      class_name) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    {
        printf("T02_getMethodNames failed!\n");
        process_error(env_ptr);
    }
}
else
```

## TO2\_getMethodNames

```
{  
    printf("TO2_getMethodNames successful!\n");  
}
```

## Related Information

"TO2\_getClassNames—Convert EBCDIC Class Name to Index" on page 1262.

## T02\_getNumberOfRecords–Return the Number of Records Used

This function returns the number of records used.

### Format

```
#include <c$to2.h>
long T02_getNumberOfRecords (const T02_PID_PTR pid_ptr,
                               T02_ENV_PTR env_ptr);
```

#### **pid\_ptr**

The pointer to the persistent identifier (PID) assigned to the collection that will be interrogated.

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The number of records used by TPF collection support (TPFCS) to hold and organize the data of the collection will be returned.

### Error Return

Not applicable.

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example prints the number of assigned records.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */
T02_ENV_PTR env_ptr;       /* PTR to the T02 environment */
T02_PID_PTR collect;       /* PTR to PID return field */
long numberofrecords;
:
:
numberofrecords = T02_getNumberOfRecords(&collect, env_ptr);

printf("The number of records is %ld.\n", numberofrecords);
```

### Related Information

None.

## TO2\_getPathInfoFor–Retrieve Path Information for a Collection Structure

This function allows the TPF collection support (TPFCS) browser to retrieve information about paths between specific structure elements of a collection or the location of a specific collection data element.

### Format

```
#include <c$to2.h>
long TO2_getPathInfoFor(const TO2_PID_PTR pid_ptr,
                          TO2_ENV_PTR env_ptr,
                          TO2_PID_PTR returnPidPtr,
                          const long *pathType,
                          const void *pathInfo);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection.

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### returnPidPtr

The pointer to where the PID assigned to the output collection will be stored.

#### pathType

The type of path being requested:

##### TO2\_PATH\_KEY

Return path information about the specified key.

##### TO2\_PATH\_RRN

Return path information about the specified relative record number (RRN).

##### TO2\_PATH\_INDEX

Return information about the location of the specified entry index.

##### TO2\_PATH\_RBA

Return information about the location of the specified relative byte address (RBA).

#### pathInfo

The pointer to the required information based on the type of path specified.

##### TO2\_PATH\_KEY

An array of three pointers where:

Pointer 1 Is the address of a string that is the key path name, or zero if the primary key path is to be used.

Pointer 2 Is the address of a long constant that contains the length of the key.

Pointer 3 Is the address of the key whose path will be retrieved.

##### TO2\_PATH\_RRN

The address of a long integer that contains the RRN whose path will be retrieved.

##### TO2\_PATH\_INDEX

The address of a long integer that contains the decimal index value whose starting location will be retrieved.

**TO2\_PATH\_RBA**

The address of a long integer that contains the hexadecimal RBA whose location will be retrieved.

**Normal Return**

The normal return is a positive value.

**Error Return**

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_UNIQUE
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_DATA_LGH
TO2_ERROR_ZERO_PID
```

**Programming Considerations**

- This function is only supported for extended collections.
- This function does not use TPF transaction services on behalf of the caller.

**Examples**

The following example retrieves the key path information for a specific key in the collection.

```
#include <c$to2.h>                /* T02 API function prototypes */
#include <stdio.h>                /* APIs for standard I/O functions */

TO2_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */
TO2_PID        keyset;
TO2_PID        returnPID;
char           key[KEYSIZE];
char           keyPath="";
long           keyLength;
long           pathType=0;
void           * pathValue[3];
:
:
/*****
/* ... now call T02 to retrieve the key path for the specified
/* key value using the specified key path.
*****/

pathType=(long) T02_PATH_KEY;      /* set path info for key */
pathValue[0]=&keyPath;           /* pointer to key path name */
pathValue[1]=&keyLength;         /* pointer to keyLength */
pathValue[2]=key;                 /* pointer to key */

if (T02_getPathInfoFor(&keyset,
                      env_ptr,
                      &returnPID,
                      &pathType,
                      pathValue) == T02_ERROR)
{
    printf("T02_getPathInfoFor failed!\n");
    process_error(env_ptr);
}
```

## T02\_getNumberOfRecords

```
}
else
{
    printf("T02_getPathInfoFor was successful!\n");
}
```

The following example retrieves the key path information for a specific RRN in the collection.

```
#include <c$to2.h>          /* T02 API function prototypes */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* Pointer to T02 Environment */
T02_PID        keyset;
T02_PID        returnPID;
long           pathType=0;
long           RRN=45637;
:
:
/*****
/* ... now call T02 to retrieve the path for the specified RRN. */
/*
*****/

pathType = (long) T02_PATH_RRN; /* set path info for RRN */

if (T02_getPathInfoFor(&keyset,
                      env_ptr,
                      &returnPID,
                      &pathType,
                      &RRN) == T02_ERROR)
{
    printf("T02_getPathInfoFor failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_getPathInfoFor was successful!\n");
}
```

## Related Information

None.



## T02\_getPIDforBrowseName—Retrieve the PID Associated with the Name

This function retrieves the persistent identifier (PID) of the given collection from the browser dictionary for the data store (DS) pointed to by the current environment pointer.

### Format

```
#include <c$to2.h>
long T02_getPIDforBrowseName (const T02_PID_PTR pid_ptr,
                                T02_ENV_PTR env_ptr,
                                const char name[]);
```

#### pid\_ptr

The pointer to where the PID assigned to the given name will be returned.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### name

The pointer to a string that is the name whose associated PID will be retrieved from the browser dictionary of the data store. The maximum name length is 32 bytes.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_DATA_LGH
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_LOCATOR_NOT_FOUND
```

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example retrieves the PID of the collection for the name COLLECTION1 from the browser dictionary for data store TEST1\_DS.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;        /* Pointer to the T02 environment */
T02_PID collectionPID;      /* named collection's returned PID */
long userToken=0;          /* no user token value */
char applid[]="APPLICATION_NAME";
char dsname[]="TEST1_DS"; /* data store name */
char name[]="COLLECTION1"; /* name to access */
:
:
if (T02_createEnv(&env_ptr,
                  &userToken,
```

## TO2\_getPIDforBrowseName

```
                                applID,  
                                dsname) == T02_ERROR)  
{  
    printf("T02_createEnv failed!\n");  
    process_error(env_ptr);  
}  
  
if (TO2_getPIDforBrowseName(&collectionPID,  
                            env_ptr,  
                            name) == T02_ERROR)  
{  
    printf("T02_getPIDforBrowseName failed!\n");  
    process_error(env_ptr);  
}  
else  
{  
    printf("T02_getPIDforBrowseName was successful!\n");  
}
```

## Related Information

- “TO2\_deleteBrowseName—Delete a Collection Name from the Browser Dictionary” on page 1248
- “TO2\_defineBrowseNameForPID—Assign a Browse Name to a Collection” on page 1246.

## TO2\_getPIDinventoryEntry–Get PID Inventory Entry for PID

This function causes TPF collection support (TPFCS) to locate the entry for the specified persistent identifier (PID) in the PID inventory collection of the data store (DS) and return a copy of the entry to the caller.

### Format

```
#include <c$to2.h>
TO2_BUF_PTR TO2_getPIDinventoryEntry (TO2_PID_PTR pid_ptr,
                                       TO2_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to a field that contains the PID with an entry that will be retrieved from the PID inventory collection of the data store.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

TPFCS returns a pointer to a formatted buffer that will contain the requested entry. The normal return is a pointer (TO2\_BUF\_PTR) to a structure (buffer) of type TO2\_BUF\_HDR (see “Type Definitions” on page 859).

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_NOT_INIT
```

### Programming Considerations

It is the responsibility of the caller to issue a freec request to release the returned buffer when the caller has finished with it.

### Examples

The following example retrieves the entry for a given PID.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

TO2_ENV_PTR    env_ptr;     /* Pointer to T02 environment */
TO2_PID        collectionPID; /* subject PID */

TO2_BUF_PTR    buffer_ptr;

TO2_ERR_CODE    to2_rc=1;    /* return code receiver */
TO2_ERR_TEXT_PTR err_textPtr; /* T02 error code text pointer */
:
{
    buffer_ptr = T02_getPIDinventoryEntry(&collectionPID,
                                         env_ptr);
    if ((long) buffer_ptr == T02_ERROR)
    {
        to2_rc = T02_getErrorCode(env_ptr);
        err_textPtr = T02_getErrorText(env_ptr, to2_rc);
        printf ("T02_getPIDinventoryEntry failed, error code - %d\n ", to2_rc);
        printf ("T02 Error Text is %s\n ", err_textPtr);
    }
}
```

## TO2\_getPIDinventoryEntry

```
    }  
    else  
        printf("TO2_getPIDinventoryEntry successful\n");  
        free buffer_ptr;  
    }
```

## Related Information

“TO2\_createPIDinventoryKey—Create PID Inventory Key from PID” on page 1244.

## TO2\_getPIDInventoryPID—Get PID of PID Inventory of the Data Store

This function returns the persistent identifier (PID) of the PID inventory for the specified data store (DS) to the specified field.

### Format

```
#include <c$to2.h>
long TO2_getPIDInventoryPID (      T02_PID_PTR pid_ptr,
                                  T02_ENV_PTR env_ptr,
                                  const char      dsname[8]);
```

#### pid\_ptr

The pointer to a field where the PID of the data store PID of the inventory will be returned.

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### dsname

A pointer to the name of the data store where the PID of the inventory PID will be returned. The character string must be 8 characters or greater in length, left-justified, and padded with blanks (X'40') on the right.

### Normal Return

The normal return is a positive value and the specified TO2\_PID\_PTR field will contain the PID of the PID inventory.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_DBID
TO2_ERROR_NOT_INIT
```

### Programming Considerations

None.

### Examples

The following example copies the PID inventory PID for data store TPFDB.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 environment */
T02_PID        inventoryPID;     /* return area for PID */

char           DSname[9]="TPFDB "; /* data store name blank padded */
T02_ERR_CODE   to2_rc=1;         /* return code receiver */
T02_ERR_TEXT_PTR err_textPtr;    /* T02 error code text pointer */
:
:
if (TO2_getPIDInventoryPID(&inventoryPID,
                          env_ptr,
                          DSname) == T02_ERROR)
{
    err_code = TO2_getErrorCode(env_ptr);
    printf ("TO2_getPIDInventoryPID failed, error code - %d\n ", to2_rc) ;
}
```

## TO2\_getPIDinventoryPID

```
        printf ("T02 Error Text is %s\n ", err_textPtr);
    }
    else
        printf("T02 get PIDinventory's PID successful\n");
```

## Related Information

- “TO2\_createDS—Create a Data Store” on page 1240
- “TO2\_getListDSnames—Retrieve the Defined Data Store Names” on page 1287.

## TO2\_getRecordAttributes—Get Attributes of a Record File Address

This function returns a temporary sequence collection that, when decoded, describes the attributes of a record file address as they pertain to TPF collection support (TPFCS).

### Format

```
#include <c$to2.h>
long TO2_getRecordAttributes (TO2_PID_PTR  attrList_ptr,
                             TO2_ENV_PTR  env_ptr,
                             void          *fileAddr_ptr);
```

#### attrList\_ptr

The pointer to where to return the persistent identifier (PID) assigned to the temporary sequence collection created in response to this request.

#### env\_ptr

The pointer to the environment as returned by the TO2\_createEnv function.

#### fileAddr\_ptr

The pointer to an 8-byte hexadecimal file address in file address reference format (FARF).

### Normal Return

The normal return is a positive value, which indicates that processing was completed normally; it does not necessarily mean that there were no errors reported in the collection for the record at that file address.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_METHOD
```

### Programming Considerations

Enter an explicit TO2\_deleteCollection function call when you have completed processing the returned temporary collection even though the temporary collection is automatically deleted when the entry control block ECB exits to ensure system resources used by that collection are cleanly released back to the system for reuse.

### Examples

The following example obtains a temporary sequence collection that, when decoded, describes the attributes of a record file address as they pertain to TPFCS.

```
#include <c$to2.h>          /* Needed for TO2 API Functions */
#include <stdio.h>          /* APIs for standard I/O functions */

TO2_PID_PTR  attrList_ptr;  /* return area for PID */
TO2_ENV_PTR  env_ptr;      /* Pointer to TO2 environment */
void         *fileAddr_ptr;
:
:
/* create an environment and set fileAddr_ptr to point to the file */
/* address of the record whose information you want to display. */
:
:
if (TO2_getRecordAttributes(attrList_ptr,
```

## T02\_getRecordAttributes

```
env_ptr,  
fileAddr_ptr) == T02_ERROR )  
{  
    printf("T02_getRecordAttributes failed!\n");  
    process_error(env_ptr);  
}  
else  
    printf("T02_getRecordAttributes successful!\n");  
    ...  
T02_deleteCollection(attrList_ptr, env_ptr);
```

## Related Information

None.



## TO2\_getRecoupIndex–Get an Index

This function is used to get the TPF collection support (TPFCS) recoup index that a persistent identifier (PID) is associated with.

### Format

```
#include <c$to2.h>
#include <c$to2r.h>
T02_BUF_PTR T02_getRecoupIndex (      T02_PID_PTR pid_ptr,
                                      T02_ENV_PTR env_ptr,
                                      const void *indexName);
```

#### **pid\_ptr**

The pointer to the PID of the recoup index. This is a return value.

#### **env\_ptr**

The pointer to the environment as returned by the T02\_createEnv function.

#### **indexName**

The pointer to an 8-byte field where the index name is to be returned.

### Normal Return

The normal return is a sequence collection. When there is a normal return, it contains a copy of the data requested in a TO2\_RCPIDX structure and is formatted as follows:

#### **TEMPL\_HDR**

The template header.

#### **TEMPL\_NAME**

The recoup index name.

#### **TEMPL\_PID**

The PID of the index collection.

#### **TEMPL\_TYPE**

TO2\_RECOUP\_TYPE:

##### **TO2\_RECOUP\_HOMOGENEOUS**

A homogeneous collection.

##### **TO2\_RECOUP\_HETEROGENEOUS**

A heterogeneous collection.

#### **TEMPL\_CNTRL**

TO2\_RECOUP\_CONTROL:

##### **TO2\_RECOUP\_CONTROL\_RESTRICT**

General entry control blocks (ECBs) for all associated PIDs.

##### **TO2\_RECOUP\_CONTROL\_USER**

User exit for the anchor PID.

#### **TEMPL\_SPARE**

Spare bytes.

#### **TEMPL\_PNAME**

User exit program name.

#### **TEMPL\_FNAME**

User exit function name.

#### **TEMPL\_TEXT**

Descriptor text.

## TO2\_getRecoupIndex

When there is a normal return, it contains a copy of the elements of the collection whose PID is returned in a TO2\_RCPIDX\_ENTRY structure and is formatted as follows:

### TEMPL\_ENTRY

The header file.

### TEMPL\_E\_TOK

The token value of the entry.

### TEMPL\_E\_TYPE

TO2\_RECOUP\_ENTRY\_TYPE:

### TO2\_RECOUP\_ENTRY\_PID

The PID.

### TO2\_RECOUP\_ENTRY\_FA

The file address.

### TEMPL\_E\_ACC

TO2\_RECOUP\_ENTRY\_ACCESS:

### TO2\_RECOUP\_ACCESS\_KEY

The access key.

### TO2\_RECOUP\_ACCESS\_INDEX

The access index.

### TEMPL\_E\_RSRV

Is reserved.

### TEMPL\_E\_DSP

Is the displacement.

### TEMPL\_E\_IDX

Is the index.

### TEMPL\_E\_LEN

Is the key length.

### TEMPL\_E\_KEY

Is the key.

## Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

TO2\_ERROR\_ENV

## Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example adds an entry to an index.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;     /* Pointer to T02 environment */

T02_PID        collect;     /*
```

```

void          *indexName;          /* index key          */
:
:
if (TO2_getRecoupIndex(&collect,
                      env_ptr,
                      indexName) == T02_ERROR)
{
    process_error()
    err_code = T02_getErrorCode(env_ptr);
    printf ("TO2_getRecoupIndex failed, error code - %d\n ", to2_rc);
    printf ("TO2 Error Text is %s\n ", err_textPtr);
}
else
    printf("Index name is %s\n", indexName);

```

## Related Information

- “TO2\_associateRecoupIndexWithPID—Create a PID to Index Association” on page 899
- “TO2\_createRecoupIndex—Create a Recoup Index” on page 998
- “TO2\_createKeySortedSetWithOptions—Create an Empty Key Sorted Set Collection” on page 982
- “TO2\_removeRecoupIndexFromPID—Remove a PID to Index Association” on page 1086
- “TO2\_getRecoupIndexForPID—Retrieve Associated Recoup Index Name” on page 1310.

## T02\_getRecoupIndexForPID—Retrieve Associated Recoup Index Name

This function is used to retrieve the name of a recoup index.

### Format

```
#include <c$to2.h>
long T02_getRecoupIndexForPID (const T02_PID_PTR  pid_ptr
                                T02_ENV_PTR    env_ptr,
                                void           *indexName);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) of a collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### indexName

The pointer to an 8-byte field that is used to return the index name. **indexName** contains the name on return.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_PID
```

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example retrieves the key to an index.

```
#include <c$to2.h>           /* Needed for T02 API Functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_PID      collect;       /* target object's PID */
T02_ENV_PTR  env_ptr;       /* Pointer to T02 environment */
T02_BUF_PTR  buf_ptr;       /* Pointer to a buffer area */

char          *indexName;    /* index name */

:
memcpy(collect, buf_ptr->data, 32);

if (T02_getRecoupIndexForPID(&collect,
                             env_ptr,
                             indexName) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    printf ("T02_getRecoupIndexForPID failed!\n");
    printf ("T02 Error Text is %s\n ", err_textPtr);
}
else
    printf("Recoup index name = %8s\n", indexName);
```

**Related Information**

- “TO2\_associateRecoupIndexWithPID—Create a PID to Index Association” on page 899
- “TO2\_createKeySortedSetWithOptions—Create an Empty Key Sorted Set Collection” on page 982
- “TO2\_createRecoupIndex—Create a Recoup Index” on page 998
- “TO2\_removeRecoupIndexFromPID—Remove a PID to Index Association” on page 1086
- “TO2\_getRecoupIndex—Get an Index” on page 1307.

## T02\_getUserAttributes–Get User Attributes

This function returns a sequence collection of the EBCDIC names of the attributes and their current values of the specified user collection.

### Format

```
#include <c$to2.h>
long T02_getUserAttributes (T02_PID_PTR pid_ptr,
                           T02_ENV_PTR env_ptr,
                           long *userToken);
```

#### pid\_ptr

The pointer to where to return the temporary persistent identifier (PID) assigned to the sequence collection of attributes.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### userToken

Reserved for future IBM use. This must point to a word that contains zeros.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

None.

### Examples

The following example retrieves the PID of the TPF dictionary.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;        /* Pointer to T02 environment */
T02_PID_PTR collect;
long      *usertoken;
:
:
if (T02_getUserAttributes(&collect,
                        env_ptr,
                        &usertoken) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    {
        printf("T02_getUserAttributes failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_getUserAttributes;
}
}

return T02_ERROR;
```

## **Related Information**

- “TO2\_getClassNames—Convert EBCDIC Class Name to Index” on page 1262
- “TO2\_getCollectionAttributes—Get the Attributes of the Collection” on page 1266.

## T02\_isDDdefined—Test If DD Name Is Defined

This function tests the specified data store (DS) for the given data definition (DD) name.

### Format

```
#include <c$to2.h>
long T02_isDDdefined (      T02_ENV_PTR  env_ptr,
                           const char    *ddname,
                           const char    *dsname);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### ddname

The pointer to a character string, which is the name of the data definition to be tested. The character string is assumed to be 32 characters in length. The name should be left-justified and padded with blanks (X'40') on the right if it less than 32 characters.

#### dsname

The pointer to a character string, which is the name of the data store that owns the data definition name (**ddname**). The character string must be 8 characters or greater in length, left-justified, and padded with blanks (X'40') on the right.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

- T02\_ERROR\_DD\_NOT\_FOUND
- T02\_ERROR\_DBID
- T02\_ERROR\_NOT\_INIT

### Programming Considerations

None.

### Examples

The following example tests the specified data definition name to determine if it has already been created.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */
{
    long      err_code=0;
    T02_ENV_PTR env_ptr=0;        /* Pointer to T02 environment */
    char      ddname[]="TEST1_DDNAME_FOR_TEST1_DS";
    char      dsname[]="TEST1_DS";
    :
    if (T02_isDDdefined(env_ptr,
                        DDname,
                        dsname) == T02_ERROR)
    {
        err_code = T02_getErrorCode(env_ptr);
    }
}
```



```
    if (err_code == 0)
        printf("%s does not exist\n",ddname);
    else
        process_error(env_ptr);
}
else
    printf("%s already exists\n",ddname);
}
```

## Related Information

- “TO2\_changeDD—Change a Data Definition” on page 1230
- “TO2\_deleteDD—Delete a Data Definition” on page 1250
- “TO2\_getDDAttributes—Get the Current DD Attribute Values” on page 1274
- “TO2\_getListDDnames—Retrieve the Defined Data Definition Names” on page 1283.

## TO2\_isExtended–Internal Format of Collections

This function determines whether TPF collection support (TPFCS) has stored the collection on the TPF database in EXTENDED format or COMPACT format.

### Format

```
#include <c$to2.h>
BOOL TO2_isExtended (const T02_PID_PTR pid_ptr,
                     T02_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection to test.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

A value of TO2\_IS\_TRUE is returned if the collection is stored on the TPF database in EXTENDED format. If the collection is not stored in EXTENDED format, a value of TO2\_IS\_FALSE is returned.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_PID
```

## Programming Considerations

- For the TO2\_IS\_FALSE result, check the TPFCS error code by entering the T02\_getErrorCode function to distinguish this result from an error return indication. If the error code is zero, the Boolean result is false. Otherwise, an error is indicated and you can retrieve the error text by entering the T02\_getErrorText function.
- This function does not use TPF transaction services on behalf of the caller.

## Examples

The following example determines if the collection resides on the TPF database in EXTENDED format.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;    /* PTR to the T02 environment */
T02_PID        collect;    /* PTR to PID return field */

:
:
if ((TO2_isExtended(&collect, env_ptr)) == TO2_IS_FALSE)
{
    printf("This is not an EXTENDED collection! \n");
}
else
{
    printf("This is an EXTENDED collection! \n");
}
```

## Related Information

None.

## T02\_migrateCollection—Migrate a Collection

This function migrates the specified collection by re-creating the collection and copying the contents of the old collection to the new collection one element at a time. The persistent identifier (PID) of the new collection is returned.

### Format

```
#include <c$to2.h>
long T02_migrateCollection (T02_PID_PTR  pid_ptr,
                           T02_ENV_PTR  env_ptr,
                           T02_PID_PTR  rpid_ptr);
```

#### pid\_ptr

A pointer to the PID assigned to the collection that will be migrated.

#### env\_ptr

A pointer to the environment as returned by the T02\_createEnv function.

#### rpids\_ptr

A pointer to the field where the PID for the new collection will be returned.

### Normal Return

A positive value.

### Error Return

A value of zero. Use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

## Programming Considerations

- This function creates a copy of an individual TPF collection support (TPFCS) collection by using the current TPFCS collection format with new pool addresses assigned to the collection.
- A commit scope will be created for the T02\_migrateCollection request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_migrateCollection request, the scope will be rolled back.

## Examples

The following example creates a copy of a collection by using the current TPFCS collection format and new pool file addresses.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* PTR to the T02 environment */
T02_PID      source_pid;   /* PID to be migrated */
T02_PID      migrated_pid; /* Migrated PID */
:
:
if (T02_migrateCollection(&source_pid,
                        env_ptr,
                        &migrated_pid) == T02_ERROR)
{
    printf("T02_migrateCollection failed! \n");
    process_error(env_ptr);
}
else
{
    printf("T02_migrateCollection successful! \n");
}
```

## Related Information

- “TO2\_addAllFrom—Add All from Source Collection” on page 886
- “TO2\_copyCollection—Make a Persistent Copy of the Collection” on page 927
- “TO2\_createDS—Create a Data Store” on page 1240
- “TO2\_deleteDS—Delete a Data Store” on page 1252
- “TO2\_migrateDS—Migrate a Data Store” on page 1320.

See *TPF Application Programming* for more information about commit scope.

## T02\_migrateDS—Migrate a Data Store

This function migrates the TPF collection support (TPFCS) system collections in a particular data store to the current TPFCS collection format with new pool addresses assigned to the collections.

### Format

```
#include <c$to2.h>
long T02_migrateDS (T02_PID_PTR  pid_ptr,
                   T02_ENV_PTR  env_ptr,
                   const char   dsname[T02_MAX_DSNAME]);
```

#### pid\_ptr

A pointer to the field that holds the returned persistent identifier (PID) assigned to the temporary sequence collection that contains a list of the PIDs of the system collections before this function was called.

#### env\_ptr

A pointer to the environment as returned by the T02\_createEnv function.

#### dsname

A pointer to a character string, which represents the data store being migrated. The character string must be 8 characters or less in length, left-justified, and padded with blanks (X'40') on the right.

### Normal Return

A positive value. The sequence collection referred to by **pid\_ptr** contains one element per system collection. The format of the data area of the element is as follows:

```
struct T02_dsCollection_element
{
    T02_PID collectionPID[T02_MAX_PID_SIZE];
};
```

### Error Return

A value of zero. Use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

## Programming Considerations

- This function re-creates the system collections contained in the data store and copies the contents of the old system collections to the new system collections one element at a time.
- You must migrate all user collections in the data store by using the T02\_migrateCollection function call. You also must delete the list of old system collections returned by this function by using the T02\_deleteCollection function call, or otherwise process the list of old system collections returned by this function.
- Enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection even though the temporary collection is automatically deleted when the entry control block (ECB) exits. This action ensures that system resources used by that collection are cleanly released back to the system for reuse.
- A commit scope will be created for the T02\_migrateDS request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_migrateDS request, the scope will be rolled back.

## Examples

The following example migrates data store TESTX.DS.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_PID      pid;          /* Holder for temporary sequence collect */
T02_ENV_PTR  env_ptr;      /* Pointer to T02 environment */
char         dsname[T02_MAX_DSNAME]="TESTX.DS";

:
:
if (T02_migrateDS(&pid, env_ptr, dsname) == T02_ERROR)
{
    printf("T02_migrateDS failed! \n");
    process_error(env_ptr);
}
else
{
    printf("T02_migrateDS successful! \n");
}

/* Process the elements in the returned sequence collection */
```

## Related Information

- “T02\_createDS—Create a Data Store” on page 1240
- “T02\_deleteCollection—Delete a Collection” on page 1036
- “T02\_getDSAttributes—Get the Attributes of the Data Store” on page 1278
- “T02\_getDSdictPID—Get the Dictionary PID of a Data Store” on page 1212
- “T02\_getDSnameForPID—Determine the DS Name for a PID” on page 1280
- “T02\_getListDSCollections—Retrieve the Data Store System Collections” on page 1285
- “T02\_getListDSnames—Retrieve the Defined Data Store Names” on page 1287
- “T02\_migrateCollection—Migrate a Collection” on page 1318
- “T02\_recreateDS—Re-create a Data Store” on page 1330.

See *TPF Application Programming* for more information about commit scope.

## T02\_reclaimPID—Reclaim a PID

This function reclaims (unmarks) a collection that has been marked for deletion but has not actually been deleted.

### Format

```
#include <c$to2.h>
long T02_reclaimPID (const T02_PID_PTR pid_ptr,
                    T02_ENV_PTR env_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) to reclaim.

#### env\_ptr

The pointer to the environment.

### Normal Return

The normal return is a positive value. This occurs if the collection is either successfully unmarked or was not originally marked for deletion.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

## Programming Considerations

A commit scope will be created for the T02\_reclaimPID request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_reclaimPID request, the scope will be rolled back.

## Examples

The following example reclaims (unmarks) a collection that is marked for deletion.

```
#include <c$to2.h>          /* T02 API function prototypes */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_PID    collect;
T02_ENV_PTR env_ptr;       /* Pointer to T02 environment */
:
:
/*****
/* Reclaim (unmark) a collection marked for deletion.
*****/
if (T02_reclaimPID(&collect,
                  env_ptr) == T02_ERROR)
{
    {
        printf("T02_reclaimPID failed!\n");
        process_error(env_ptr);
    }
}
else
{
    {
        printf("T02_reclaimPID successful!\n");
    }
}
```

## Related Information

“T02\_deletePID—Delete a Persistent Identifier and Its Backing Store” on page 1352.



## TO2\_reconstructCollection–Reconstruct Collection

This function reconstructs either the data chains or the internal mapping of a collection that has errors. A reconstruction report that contains a list of the actions taken on the collection along with any errors that cannot be resolved is returned. The errors that cannot be resolved must be corrected manually by the database administrator (DBA).

### Format

```
#include <c$to2.h>
long TO2_reconstructCollection (      T02_PID_PTR      collectPtr,
                                     T02_ENV_PTR      env_ptr,
                                     const T02_OPTION_PTR optionListPtr,
                                     T02_PID_PTR      reconReportPtr,
                                     enum T02_RECON_METHOD reconMethod);
```

#### **collectPtr**

The pointer to the persistent identifier (PID) assigned to the collection that will be reconstructed.

#### **env\_ptr**

The pointer to the environment as returned by the TO2\_createEnv function.

#### **optionListPtr**

The pointer to a return options list from a TO2\_createOptionList call or NULL if no options are required.

#### **reconReportPtr**

The pointer to where to return the temporary PID assigned to the sorted bag collection that represents the reconstruction report.

#### **reconMethod**

Specify one of the following:

##### **TO2\_RECON\_DATA**

Use this term to specify that the data chains of the collection will be rebuilt from the internal mapping of the collection.

##### **TO2\_RECON\_KEYS**

Use this term to specify that the key (index) internal mapping of the collection will be rebuilt from the data chains of the collection.

##### **TO2\_RECON\_DIRECT**

Use this term to specify that the directory internal mapping of the collection will be rebuilt from the data and key chains of the collection.

### Normal Return

The normal return is a positive value, which indicates that processing was completed normally; it does not necessarily mean that the collection is free of errors.

### Error Return

An error return is indicated by a zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_METHOD
```

### Programming Considerations

- A commit scope will be created for the reconstructCollection request. If the request is successful, the scope will be committed. If reconstruction processing is unable to be completed normally, the scope will be rolled back.
- Use the T02\_validateCollection function after the T02\_reconstructCollection function is returned to determine if any additional errors exist in the collection.

### Examples

The following example reconstructs the data chain of a collection. The example assumes that the data chain of the collection is corrupted, but the internal mapping is still intact.

```
#include <c$to2.h>          /* Needed for T02 API Functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;
T02_PID        collect;
T02_PID        reconReport;
:
:
/*****
/* Reconstruct the collection's data chain using the internal
/* directory structure.
*****/
if (T02_reconstructCollection(&collect,
                             env_ptr,
                             NULL,
                             &reconReport,
                             T02_RECON_DATA) == T02_ERROR)
{
    printf("T02_reconstructCollection failed!\n");
    process_error(env_ptr);
}
else
    printf("T02_reconstructCollection successful!\n");
```

### Related Information

“T02\_validateCollection—Cause a Collection to Be Validated” on page 1340.

## T02\_recoupCollection–Return Head of Chain File Addresses

This function returns a sequence collection containing the head of chain file address for the collection specified for use by recoup.

### Format

```
#include <c$to2.h>
long T02_recoupCollection (const T02_PID_PTR  pid_ptr,
                              T02_ENV_PTR   env_ptr,
                              T02_PID_PTR   rpid_ptr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) of the collection that will be recouped.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### rpidd\_ptr

The pointer to the field to hold the temporary PID of the returned sequence collection.

### Normal Return

The sequence collection contains one 8-byte element per pool file chain for the target collection. The format of this element (representing the head of the file chain) is as follows:

```
struct chainElement
{
    u_char    recordID[2];
    u_char    reserved[2];
    u_char    file_Address[8];
}
```

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

T02\_ERROR\_RECOUP\_ABORT

### Programming Considerations

When the caller is done with the returned collection, delete the collection by using the T02\_deleteCollection function.

### Examples

The following example recoups the specified collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr;       /* Pointer to T02 environment */
T02_PID_PTR collect;
T02_PID_PTR returnPidPtr;
:
:
if (T02_recoupCollection(&collect,
                        env_ptr,
                        &returnPidPtr) == T02_ERROR)
```

## TO2\_recoupCollection

```
{
    printf("TO2_recoupCollection failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_recoupCollection successful!\n");
}
```

## Related Information

- “TO2\_deleteCollection–Delete a Collection” on page 1036
- “TO2\_getRecoupIndex–Get an Index” on page 1307
- “TO2\_getRecoupIndexForPID–Retrieve Associated Recoup Index Name” on page 1310
- “TO2\_recoupDS–Return a List of Internal PIDs to Be Recouped” on page 1327.

## TO2\_recoupDS—Return a List of Internal PIDs to Be Recouped

This function returns a sequence collection that contains a list of the TPF collection support (TPFCS) internal persistent identifiers (PIDs) that need to be recouped.

### Format

```
#include <c$to2.h>
long  T02_recoupDS(const T02_PID_PTR  pid_ptr,
                      T02_ENV_PTR    env_ptr,
                      char            dsname)
```

#### pid\_ptr

The pointer to the location where the persistent identifier (PID) of the returned collection is to be stored.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### dsname

The pointer to a character string, which is the name of the data store (DS) containing the collections to be recouped. The character string must be 8 characters in length, left-justified, and padded with blanks (X'40') on the right.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

- Each element in the sequence collection returned contains a 32-byte PID starting at displacement 0.
- When the caller is done with the returned collection, delete the collection by using the T02\_deleteCollection function.

### Examples

The following example retrieves the sequence collection of internal PIDs for the data store indicated.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_PID    collect;
char       dsname[] = "MYDS "; /* Data store name */
T02_ENV_PTR env_ptr;          /* Pointer to T02 environment */
:
if (T02_recoupDS(&collect,
                env_ptr,
                dsname) == T02_ERROR)

{
    printf("T02_recoupDS failed!\n");
    process_error(env_ptr);
}
```

## TO2\_recoupDS

```
else
{
    printf("TO2_recoupDS successful!\n");
}
```

## Related Information

- “TO2\_recoupCollection—Return Head of Chain File Addresses” on page 1325
- “TO2\_getRecoupIndex—Get an Index” on page 1307
- “TO2\_getRecoupIndexForPID—Retrieve Associated Recoup Index Name” on page 1310.

## TO2\_recoupPT—Clear the Pool Reuse Table

This function clears the TPF collection support (TPFCS) in-core pool reuse table at the location referenced by CINFC tag CMMTO22.

### Format

```
#include <c$to2.h>
long TO2_recoupPT (TO2_ENV_PTR env_ptr)
```

#### **env\_ptr**

The pointer to the environment as returned by the TO2\_createEnv function.

### Normal Return

A positive value.

### Error Return

A value of zero. When zero is returned, use the TO2\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

## Programming Considerations

None.

## Examples

The following example clears the TPF collection support (TPFCS) pool reuse table on the local processor.

```
#include <c$to2.h>          /* Needed for TO2 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

TO2_PID    collect;
TO2_ENV_PTR env_ptr;      /* Pointer to TO2 environment */
:
:
if (TO2_recoupPT (env_ptr) == TO2_ERROR)
{
    printf("TO2_recoupPT failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_recoupPT successful!\n");
}
```

## Related Information

- “TO2\_recoupCollection—Return Head of Chain File Addresses” on page 1325
- “TO2\_recoupDS—Return a List of Internal PIDs to Be Recouped” on page 1327
- “TO2\_getRecoupIndex—Get an Index” on page 1307
- “TO2\_getRecoupIndexForPID—Retrieve Associated Recoup Index Name” on page 1310.

## T02\_recreateDS—Re-create a Data Store

This function re-creates a data store in an initialized state.

### Format

```
#include <c$to2.h>
long T02_recreateDS (T02_PID_PTR pid_ptr,
                    T02_ENV_PTR env_ptr,
                    const char dsname[T02_MAX_DSNAME]);
```

#### pid\_ptr

A pointer to the field that holds the returned persistent identifier (PID) assigned to the temporary sequence collection that contains a list of the PIDs of the system collections before this function was called.

#### env\_ptr

A pointer to the environment as returned by the T02\_createEnv function.

#### dsname

A pointer to a character string, which represents the data store being re-created. The character string must be 8 characters or less in length, left-justified, and padded with blanks (X'40') on the right.

### Normal Return

A positive value. The sequence collection referred to by **pid\_ptr** contains one element per system collection. The format of the data area of the element is as follows:

```
struct T02_dsCollection_element
{
    T02_PID collectionPID[T02_MAX_PID_SIZE];
};
```

### Error Return

A value of zero. Use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

- This function re-creates and replaces all system collections contained in the data store using new pool file addresses.
- You must delete the list of old system collections returned by this function by using the T02\_deleteCollection function call or otherwise process the list of old system collections returned by this function.
- If you do not delete all user collections and recoup indexes, the pool records associated with those collections will not be available for use until the next recoup function is run.
- You cannot re-create the TPFDB data store.
- Enter an explicit T02\_deleteCollection function call when you have completed processing the returned temporary collection even though the temporary collection is automatically deleted when the entry control block (ECB) exits. This action ensures that system resources used by that collection are cleanly released back to the system for reuse.
- A commit scope will be created for the T02\_recreateDS request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_recreateDS request, the scope will be rolled back.



## Examples

The following example re-creates data store TESTX.DS.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_PID      pid;           /* Holder for temp sequence collect */
T02_ENV_PTR  env_ptr;       /* Pointer to T02 environment */
char         dsname[T02_MAX_DSNAME]="TESTX.DS";
:
:
if (T02_recreateDS(&pid, env_ptr, dsname) == T02_ERROR)
{
    printf("T02_recreateDS failed! \n");
    process_error(env_ptr);
}
else
{
    printf("T02_recreateDS successful! \n");
}

/* Process the elements in the returned sequence collection */
```

## Related Information

- “TO2\_createDS—Create a Data Store” on page 1240
- “TO2\_getDSAttributes—Get the Attributes of the Data Store” on page 1278
- “TO2\_getDSdictPID—Get the Dictionary PID of a Data Store” on page 1212
- “TO2\_getDSnameForPID—Determine the DS Name for a PID” on page 1280
- “TO2\_getListDSCollections—Retrieve the Data Store System Collections” on page 1285
- “TO2\_getListDSnames—Retrieve the Defined Data Store Names” on page 1287
- “TO2\_migrateDS—Migrate a Data Store” on page 1320.

See *TPF Application Programming* for more information about commit scope. See *TPF Database Reference* for more information about data stores.

## T02\_removeBrowseKey—Remove the Element Using the Specified Key

This function locates the entry represented by the specified key and removes the entry from the dictionary. The passed key string is assumed to be 64 bytes long.

### Format

```
#include <c$to2.h>
long T02_removeBrowseKey (      T02_ENV_PTR  env_ptr,
                              const void    *key);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### key

The pointer to the EBCDIC key to use to locate the dictionary entry. The passed key string is assumed to be 64 bytes long.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

### Programming Considerations

A commit scope will be created for the T02\_removeBrowseKey request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_removeBrowseKey request, the scope will be rolled back.

### Examples

The following example removes the entry associated with the key provided from the dictionary.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;        /* Pointer to T02 environment */
char      *key;
:
:
if (T02_removeBrowseKey(env_ptr,
                        key) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    {
        printf("T02_removeBrowseKey failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_removeBrowseKey successful!\n");
}
}
```

### Related Information

- “T02\_atBrowseKey—Retrieve the Element Using the Specified Key from the Browser Dictionary” on page 1224

## **TO2\_removeBrowseKey**

- “TO2\_atBrowseKeyPut—Replace the Element Using the Specified Key from the Browser Dictionary” on page 1226
- “TO2\_atBrowseNewKeyPut—Store the Element Using the Specified Key in the Browser Dictionary” on page 1228
- “TO2\_getBrowseDictPID—Get the Browser Dictionary PID” on page 1254.

## T02\_restart–Restart TPF Collection Support at Cycle-Up

This function is used by the cycle-up program CJ04 to request a TPFCS restart.

### Format

```
#include <c$to2.h>
long T02_restart (T02_ENV_PTR env_ptr,
                  long         *cycleType);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### cycleType

This parameter is reserved for future use; code a 0 for now.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

T02\_ERROR\_NOT\_INIT

### Programming Considerations

None.

### Examples

The following example restarts TPFCS services.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;        /* Pointer to T02 environment */
long      cycleType;
:
:
if (T02_restart(env_ptr,
                &cycleType) == T02_ERROR)
{
    err_code = T02_getErrorCode(env_ptr);
    {
        printf("T02_restart failed!\n");
        process_error(env_ptr);
    }
}
else
{
    printf("T02_restart successful!\n");
}
}
```

### Related Information

None.

## TO2\_setGetTextDump–Set Text Dump On or Off

This function sets the TPF collection support (TPFCS) text dump function on or off. TextDump is an OPR dump that TPFCS will issue on any call to the T02\_getErrorText function. Normally, application programs will only issue a T02\_getErrorText request when the program has received an unexpected error code from a TPFCS request. Turning on the T02\_setGetTextDump function allows a dump to be taken and the problem resolved. The state of the T02\_setGetTextDump attribute is saved and restored after every IPL.

### Format

```
#include <c$to2.h>
long  T02_setGetTextDump (T02_ENV_PTR  env_ptr,
                          char          *state[]);
```

#### env\_Ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### state

A pointer to a character string that contains one of the following values:

##### ON

Set text dump on.

##### OFF

Set text dump off.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_NOT_INIT
TO2_ERROR_PARAM
```

## Programming Considerations

By issuing a T02\_getClassAttributes request for class, TPFCS will retrieve the current attribute value for the TextDump function.

### Examples

The following example sets the TextDump attribute state to ON.

```
#include <c$to2.h>                /* Needed for T02 API Functions */
#include <stdio.h>                 /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 environment */

char           textDumpOn[3]="ON"; /* value to turn text dump on */
T02_ERR_CODE   to2_rc=1;         /* return code receiver */
T02_ERR_TEXT_PTR err_textPtr;    /* T02 error code text pointer */
:
{
if ((to2_rc = T02_setGetTextDump(env_ptr,
```

## T02\_setGetTextDump

```
                                textDumpOn)) == T02_ERROR)
{
    to2_rc = T02_getErrorCode(env_ptr);
    err_textPtr = T02_getErrorText(env_ptr, to2_rc);
    printf ("T02_setGetTextDump failed, error code - %d\n ", to2_rc);
    printf ("T02 Error Text is %s\n ", err_textPtr);
}
else
    printf("T02 Text Dump turned ON\n");
}
```

The following example sets the TextDump attribute state to OFF.

```
#include <c$to2.h>                                /* Needed for T02 API Functions */

T02_ENV_PTR    env_ptr;                            /* Pointer to T02 environment */

char           textDumpOff[4]="OFF"; /* turn text dump off */
T02_ERR_CODE   to2_rc=1; /* return code receiver */
T02_ERR_TEXT_PTR err_textPtr; /* T02 error code text pointer */
:
{
    if ((to2_rc = T02_setGetTextDump(env_ptr,
                                    textDumpOff)) == T02_ERROR)
    {
        to2_rc = T02_getErrorCode(env_ptr);
        err_textPtr = T02_getErrorText(env_ptr, to2_rc);
        printf ("T02_setGetTextDump failed, error code - %d\n ", to2_rc);
        printf ("T02 Error Text is %s\n ", err_textPtr);
    }
    else
        printf("T02 Text Dump turned OFF\n");
}
```

## Related Information

- “T02\_getClassAttributes—Get Attributes of a Class” on page 1256
- “T02\_setMethodTrace—Set Method Trace On or Off” on page 1337.

## T02\_setMethodTrace—Set Method Trace On or Off

This function sets the TPF collection support (TPFCS) method trace function on or off. The state of the method trace is saved and restored after every IPL.

### Format

```
#include <c$to2.h>
long T02_setMethodTrace (T02_ENV_PTR env_ptr,
                        char *state[]);
```

#### env\_Ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### state

A pointer to a character string that contains one of the following values:

##### ON

Set method trace on.

##### OFF

Set method trace off.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_NOT_INIT
T02_ERROR_PARAMETER
```

## Programming Considerations

By issuing a T02\_getClassAttributes request for class, TPFCS will retrieve the current attribute value for the method trace.

### Examples

The following example sets the method trace attribute state to ON.

```
#include <c$to2.h> /* Needed for T02 API Functions */
#include <stdio.h> /* APIs for standard I/O functions */

T02_ENV_PTR env_ptr; /* Pointer to T02 environment */

char methodTraceOn[3]="ON"; /* turn method trace on */
T02_ERR_CODE to2_rc=1; /* return code receiver */
T02_ERR_TEXT_PTR err_textPtr; /* T02 error code text pointer */
:
{
if ((to2_rc = T02_setMethodTrace(env_ptr,
                                methodTraceOn)) == T02_ERROR)
{
to2_rc = T02_getErrorCode(env_ptr);
err_textPtr = T02_getErrorText(env_ptr, to2_rc);
printf ("T02_setMethodTrace failed, error code - %d\n ", to2_rc);
printf ("T02 Error Text is %s\n ", err_textPtr);
}
```

## TO2\_setMethodTrace

```
}  
else  
    printf("T02 Method Trace turned ON\n");  
}
```

The following example sets the method trace attribute state to OFF.

```
#include <c$to2.h>                                /* Needed for T02 API Functions */  
  
T02_ENV_PTR    env_ptr;                          /* Pointer to T02 environment */  
  
char           methodTraceOff[4]="OFF"; /* turn method trace off */  
T02_ERR_CODE   to2_rc=1;                      /* return code receiver */  
T02_ERR_TEXT_PTR err_textPtr;                 /* T02 error code text pointer */  
:  
{  
    if ((to2_rc = T02_setMethodTrace(env_ptr,  
                                     methodTraceOff)) == T02_ERROR)  
    {  
        to2_rc = T02_getErrorCode(env_ptr);  
        err_textPtr = T02_getErrorText(env_ptr, to2_rc);  
        printf ("T02_setMethodTrace failed, error code - %d\n ", to2_rc);  
        printf ("T02 Error Text is %s\n ", err_textPtr);  
    }  
    else  
        printf("T02 Method Trace turned OFF\n");  
}
```

## Related Information

- “TO2\_getClassAttributes–Get Attributes of a Class” on page 1256
- “TO2\_setGetTextDump–Set Text Dump On or Off” on page 1335.



## T02\_taskDispatch—Dispatch a TPF Collection Support Task

This function is an internal application programming interface (API) that allows TPF collection support (TPFCS) to dispatch requests to the TPF system and receive control back at a later time.

**Note:** This function must not be used by application programs.

### Format

```
#include <c$to2.h>
void T02_taskDispatch (T02_ENV_PTR env_ptr,
                      void *task);
```

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### task

The pointer to a storage area that contains a task.

### Normal Return

Void.

### Error Return

Not applicable.

### Programming Considerations

None.

### Examples

The following example activates the task indicated.

```
#include <c$to2.h>                /* Needed for T02 API functions */

T02_ENV_PTR env_ptr;              /* Pointer to T02 environment */
void *task;
:
T02_taskDispatch(env_ptr,
                 task);
```

### Related Information

None.

## T02\_validateCollection—Cause a Collection to Be Validated

This function validates the specified structure of the collection and returns a collection of error messages for any errors found.

### Format

```
#include <c$to2.h>
long T02_validateCollection (const T02_PID_PTR pid_ptr,
                              T02_ENV_PTR env_ptr,
                              T02_PID_PTR validReportPtr);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection that will be validated.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### validReportPtr

The pointer to where to return the temporary PID assigned to the sorted bag collection that represents the validation error report.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_METHOD
```

## Programming Considerations

A commit scope will be created for the T02\_validateCollection request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_validateCollection request, the scope will be rolled back.

### Examples

The following example verifies the PID indicated and returns the error collection PID if applicable.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

long      err_code;
T02_ENV_PTR env_ptr;      /* Pointer to T02 environment */
T02_PID_PTR collect,
T02_PID_PTR report_PidPtr;
:
:
if (T02_validateCollection(&collect,
                          env_ptr,
                          report_PidPtr) == T02_ERROR)
{
    printf("T02_validateCollection failed!\n");
    process_error(env_ptr);
}
```

```
else
{
    printf("TO2_validateCollection successful!\n");
}
```

## **Related Information**

"TO2\_reconstructCollection–Reconstruct Collection" on page 1323.

## T02\_validateKeyPath—Cause a Key Path to Be Validated

This function validates the structure of the specified key path and forces a TPF collection support (TPFCS) dump on the first error found. Once an error is found, this function will stop the validation and return to the caller.

### Format

```
#include <c$to2.h>
long T02_validateKeyPath(const T02_PID_PTR  pid_ptr,
                             T02_ENV_PTR   env_ptr,
                             T02_PID_PTR   validReportPtr,
                             const char    *name);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) assigned to the collection whose key path will be validated.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### validReportPtr

The pointer to where to return the temporary PID assigned to the sorted bag collection that represents the validation report.

#### name

The pointer to a string that is the name assigned to the key path that will be validated.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_ENV
TO2_ERROR_LOCATOR_NOT_FOUND
TO2_ERROR_METHOD
TO2_ERROR_NOT_INIT
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

- This application programming interface (API) will cause a storage dump on the first error found and then return to the caller. The report collection will be empty. The following shows the dump and contains the index records that caused the validation to fail. Examine the index records to determine what action to take to rebuild the key path structure by using either the ZAFIL command or the ZBROW COLLECTION command with the RECONSTRUCT parameter specified.

```
CPSE0052E 08.41.44 IS-0001 SS-BSS SSU-HPN SE-NODUMP OPR-I0200E6
010000C CJ00 00000000
CLASS-NDXDATAPAGE METHOD-NDXDPGI8,validateIndex
T02 STRUCTURE LOGIC ERROR. DETAIL CODE 00001
```

- Use index name TO2\_PRIME\_KEYPATH to cause the validation to be performed on the primary key path.

## Examples

The following example validates the primary key path.

```
#include <c$to2.h>                /* T02 API function prototypes */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */
T02_PID        keyset;
T02_PID        errorReport;
char           keyPathName='T02_PRIME_KEYPATH'; /* validate primary */

:
if (T02_validateKeyPath(&keyset,
                        env_ptr,
                        &errorReport,
                        keyPathName) == T02_ERROR)
{
    printf("T02_validateKeyPath failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_validateKeyPath was successful!\n");
}
```

## Related Information

None.

**TO2\_validateCollection**

---

## **TPF Collection Support: Independent APIs**

This chapter describes the TPF collection support (TPFCS) independent application program interfaces (APIs) that are common across all collection types.

## T02\_class–Retrieve the User Class ID of the Collection

This function is used to retrieve the user class identifier (ID) as set by the T02\_setClass function. If the ID has not been set, a null string is returned to the **field** parameter.

### Format

```
#include <c$to2.h>
long T02_class (const T02_PID_PTR   pid_ptr,
                  T02_ENV_PTR     env_ptr,
                  void             *field);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) of the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### field

The pointer to an area in which the 32-byte class ID value will be returned.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero return code. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_PID
```

## Programming Considerations

A commit scope will be created for the T02\_class request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_class request, the scope will be rolled back.

### Examples

The following example retrieves the user class ID value set for a collection.

```
#include <c$to2.h>           /* Needed for T02 API functions */
#include <stdio.h>           /* APIs for standard I/O functions */

T02_ENV_PTR   env_ptr;      /* Pointer to T02 Environment */
T02_PID       blob;
char classID[32] = "          ";
:
:
if (T02_class(&blob,
             env_ptr,
             classID) == T02_ERROR)
{
    printf("T02_class failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_class successful!\n");
    printf("User Class ID value is - %s \n",classID);
}
```



```
}  
:  
if (memcmp("LOG", returned_value, 4))
```

## **Related Information**

“TO2\_setClass—Set the User Class ID of the Collection” on page 1354.

## T02\_convertBinPIDtoEBCDIC–Make a PID Printable

This function converts a binary persistent identifier (PID) to an EBCDIC string.

### Format

```
#include <c$to2.h>
long T02_convertBinPIDtoEBCDIC (const T02_PID_PTR  binary_pid_ptr,
                                   T02_ENV_PTR    env_ptr,
                                   char            EBCDIC_pid_output[64]
                                   long            *pid_output_length);
```

#### binary\_pid\_ptr

The pointer to a field that is 32 characters long and contains the binary PID that will be formatted.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### EBCDIC\_pid\_output

The pointer to a field that is 64 characters long in which the converted EBCDIC PID will be stored.

#### pid\_output\_length

The pointer to where the length of the converted PID will be stored on return. This length includes trailing zeros in the PID.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero return code. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error code is common for this function:

T02\_ERROR\_ENV

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example converts a PID to an EBCDIC string.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
T02_PID        collect;
char           pid_buffer[64];
long           pid_length;
:
:
/*****
/* Make sure that the collection PID is assigned before arriving here.*/
*****/

pid_length = sizeof(pid_buffer);
if (T02_convertBinPIDtoEBCDIC(&collect,
                             env_ptr,
                             pid_buffer,
                             &pid_length) == T02_ERROR)
```

## TO2\_convertBinPIDtoEBCDIC

```
{
    printf("TO2_convertBinPIDtoEBCDIC failed!\n");
    process_error(env_ptr);
}
else
{
    printf("TO2_convertBinPIDtoEBCDIC successful!\n");
    printf("The PID may now be displayed as %s",pid_buffer);
}
```

## Related Information

"TO2\_convertEBCDICtoBinPID—Convert EBCDIC String to PID" on page 1350.

## T02\_convertEBCDICtoBinPID—Convert EBCDIC String to PID

This function converts an EBCDIC string persistent identifier (PID) to a binary PID.

### Format

```
#include <c$to2.h>
long T02_convertEBCDICtoBinPID(const char      EBCDIC_pid_ptr[64],
                                T02_ENV_PTR env_ptr,
                                T02_PID_PTR binary_pid_output,
                                long          *pid_output_length);
```

#### EBCDIC\_pid\_ptr

The pointer to a field that contains 64 EBCDIC characters that will be converted to a binary PID.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### binary\_pid\_output

The pointer to a field that is 32 characters long in which the converted binary PID will be stored.

#### pid\_input\_length

The pointer to the length of the EBCDIC PID to be converted. On return, this field will be overlaid with the binary length of the converted PID.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero return code. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
TO2_ERROR_DATA_LGH
TO2_ERROR_ENV
TO2_ERROR_PID
TO2_ERROR_ZERO_PID
```

### Programming Considerations

This function does not use TPF transaction services on behalf of the caller.

### Examples

The following example converts an EBCDIC string PID to a binary PID.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions*/

T02_ENV_PTR    env_ptr;          /* Pointer to the T02 environment */
T02_PID        outputPID;
char           input_PID[64];
long           pid_length;
:
:
/*****
/* Make sure that input_PID is set to the EBCDIC string PID you want */
/* to convert before this code. */
*****/
```

```
pid_length = sizeof(input_PID);

if (T02_convertEBCDICtoBinPID(inputPID,
                             env_ptr,
                             &output_PID,
                             &pid_length) == T02_ERROR)
{
    printf("T02_convertEBCDICtoBinPID failed!\n");
    process_error(env_ptr);
}
else
{
    printf("T02_convertEBCDICtoBinPID successful!\n");
}
```

## Related Information

"T02\_convertBinPIDtoEBCDIC—Make a PID Printable" on page 1348.

## T02\_deletePID—Delete a Persistent Identifier and Its Backing Store

This function deletes a persistent identifier (PID) from the database. This function is identical to the T02\_deleteCollection function.

After you enter the T02\_deletePID function, the collection is either marked for deletion or actually deleted from the database and cannot be accessed by other TPFCS functions. For persistent short-term collections and temporary collections, the delete always takes place immediately. For persistent long-term collections, the delete is controlled by the data store characteristics set with the ZOODB command. A persistent long-term collection marked for deletion can be reclaimed by entering the T02\_reclaimPID function in the time period between the T02\_deletePID request and the time the actual deletion occurs (48 hours).

### Format

```
#include <c$to2.h>
long T02_deletePID (const T02_PID_PTR  pid_ptr,
                    T02_ENV_PTR      env_ptr);
```

#### pid\_ptr

The pointer to the PID to be deleted.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero return code. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_PID
```

### Programming Considerations

A commit scope will be created for the T02\_deletePID request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_deletePID request, the scope will be rolled back.

### Examples

The following example deletes the persistent identifier (PID) representing a TPFCS database collection.

```
#include <c$to2.h>                /* Needed for T02 API functions */
#include <stdio.h>                /* APIs for standard I/O functions */

T02_ENV_PTR    env_ptr;          /* Pointer to T02 Environment */
T02_PID        blob;

T02_ERR_CODE   err_code;         /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr;   /* T02 error code text pointer */
:
if (T02_deletePID(&blob,
                  env_ptr) == T02_ERROR)
```

```
{
    printf("TO2_deletePID failed!\n");
    process_error(env_ptr);
}
else
    printf("TO2_deletePID successful!\n");
:
```

## **Related Information**

- “TO2\_deleteCollection—Delete a Collection” on page 1036
- “TO2\_reclaimPID—Reclaim a PID” on page 1322.

## T02\_setClass—Set the User Class ID of the Collection

This function assigns a user class identifier (ID) to a collection. The maximum value for the class ID is 32 bytes.

### Format

```
#include <c$to2.h>
long T02_setClass (const T02_PID_PTR  pid_ptr,
                    T02_ENV_PTR      env_ptr,
                    const void        *class);
```

#### pid\_ptr

The pointer to the persistent identifier (PID) of the collection.

#### env\_ptr

The pointer to the environment as returned by the T02\_createEnv function.

#### class

The pointer to an area that contains the class value that will be assigned to the collection.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a zero return code. When zero is returned, use the T02\_getErrorCode function to determine the specific error code. For more information, see “Error Handling” on page 860.

The following error codes are common for this function:

```
T02_ERROR_ENV
T02_ERROR_PID
```

### Programming Considerations

A commit scope will be created for the T02\_setClass request. If the request is successful, the scope will be committed. If an error occurs while processing the T02\_setClass request, the scope will be rolled back.

### Examples

The following example sets a user class ID value for a specified collection.

```
#include <c$to2.h>          /* Needed for T02 API functions */
#include <stdio.h>          /* APIs for standard I/O functions */

T02_ENV_PTR  env_ptr;      /* Pointer to T02 Environment */
T02_PID      blob;

char classID[32] = "T02.User.Class.ID";

T02_ERR_CODE  err_code;    /* T02 error code value */
T02_ERR_TEXT_PTR err_text_ptr; /* T02 error code text pointer */
:
if (T02_setClass(&blob,
                env_ptr,
                classID) == T02_ERROR)
{
    printf("T02_setClass failed!\n");
    process_error(env_ptr);
}
```



```
else  
    printf("TO2_setClass successful!\n");  
    ...
```

## **Related Information**

"TO2\_class—Retrieve the User Class ID of the Collection" on page 1346.

**TO2\_setClass**

---

## External Device Support

The TPFxd package is designed to allow an application to store data on an external device and, at some later point in time, retrieve it. It is intended to eliminate the need for any knowledge of the storage medium.

This chapter provides information about:

- Initial setup for external device support
- Writing records to an external device
- Retrieving data from an external device
- The TPFxd\_ functions.

---

### Initial Setup

To make use of this facility, there needs to be some initial setup. This involves defining the tape name to be used, defining the tape group name and selecting the tape drives. To facilitate the use of tapes with the TPFxd\_ functions, do the following:

1. Define the ARCHIVE (ACR) tape group using the ZTGRP command on any processor that is using TPFxd\_ functions.
2. Using the ZTDEV command, assign the group ARCHIVE to any tape drives that are going to be used for either input or output TPFxd\_ functions.
3. Using the ZTDEV command, enable automount on tape drives that will be used as output devices.
4. Define the ARC tape name as both input and output and unblocked, set the generation number to blanks, and assign it to the ARCHIVE tape group through the ZTLBL command.

---

### Writing Records

When an application wants to write a record or set of records to an external device, do the following:

1. Issue a TPFxd\_archiveStart request to obtain a token which will be used on all subsequent TPFxd\_ functions.
2. Allocate storage for a **TPFxd\_locationmap** (see the c\$tpxd.h header file for the size of that item).
3. Issue a TPFxd\_open request. This will update the **TPFxd\_locationmap** allocated in step 2, this information should be preserved because it will be required to retrieve the data at a later point in time.
4. Issue either a TPFxd\_write or a TPFxd\_writeBlock request, depending on whether the data is in heap storage or within a TPF data level.
5. Repeat the write requests until all the data that is intended to be together has been written.
6. Issue a TPFxd\_close request to ensure that the data has been written to the external device.
7. Repeat steps 2 through steps 6 for any additional groups of data that need to be written.
8. Issue a TPFxd\_archiveEnd request. This will close and dismount the tape that was mounted in step 3.

---

## Retrieving Data

The data has now been written to the external device and may be retrieved at a later time. To retrieve the data, follow the steps outlined below:

1. Locate the **TPFxd\_locationmap** that is associated with the data that is to be retrieved.
2. Issue a **TPFxd\_open** request (for input) and provide the **TPFxd\_locationmap** from step 1.
3. If the appropriate tape is not already mounted, a tape drive will be selected from one that was assigned to the group ARCHIVE along with a tape name that is not in use. The operator will then be prompted to mount that tape on the drive indicated using the tape name selected. Once the tape has been mounted and positioned to the start of the data that is required, control will be returned to the application that issued the **TPFxd\_open** request.
4. Issue **TPFxd\_read** or **TPFxd\_readBlock** requests until all the data has been retrieved.
5. Issue a **TPFxd\_close** request. The tape will be left in **reserved** status until it is needed by another **TPFxd\_open** request or that tape drive is needed for a different volume serial number (VOLSER), in which case the tape will be closed and dismounted.

The data that was written to the external device is still there and can be retrieved again and again by following the same set of steps. If several groups of data are to be retrieved, request the positioning strings in ascending order to allow for the most effective record retrieval.

---

## Error Codes

The following table shows the external device functions and error codes that are applicable to each function. Refer to the `c$tpxd.h` header file for detailed descriptions of each error.

Refer to the following key for the function names.

**Key:**

<b>A</b>	<b>TPFxd_archiveStart</b>
<b>B</b>	<b>TPFxd_open</b>
<b>C</b>	<b>TPFxd_nextVolume</b>
<b>D</b>	<b>TPFxd_getPosition</b>
<b>E</b>	<b>TPFxd_getPrevPosition</b>
<b>F</b>	<b>TPFxd_getVOLSER</b>
<b>G</b>	<b>TPFxd_getVOLSERlist</b>
<b>H</b>	<b>TPFxd_setPosition</b>
<b>I</b>	<b>TPFxd_sync</b>
<b>J</b>	<b>TPFxd_write</b>
<b>K</b>	<b>TPFxd_read</b>
<b>L</b>	<b>TPFxd_writeBlock</b>
<b>M</b>	<b>TPFxd_readBlock</b>

**N**      TPFxd\_close  
**O**      TPFxd\_archiveEnd

Error Code	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
TPFxd_ERROR_archiveUnavail (-1 )				X	X			X	X	X	X	X	X	X	X
TPFxd_ERROR_levelInUse (-2 )													X		
TPFxd_ERROR_maxArchive (-3 )	X	X	X												
TPFxd_ERROR_notOpen (-4 )				X	X			X	X	X	X	X	X	X	
TPFxd_ERROR_noDDN (-5 )															
TPFxd_ERROR_noPosition (-6 )		X	X	X	X	X	X	X							
TPFxd_ERROR_positionIncorrect (-7)						X	X	X							
TPFxd_ERROR_readOnly (-8 )		X								X		X			
TPFxd_ERROR_recordTooLong (-9 )										X	X		X		
TPFxd_ERROR_recordTooShort (-10)											X		X		
TPFxd_ERROR_timeOut (-11)		X	X												
TPFxd_ERROR_volserMismatch (-12)		X	X					X							
TPFxd_ERROR_EOVwarning (-13)									X	X		X		X	
TPFxd_ERROR_HWerror (-14)		X	X	X	X			X	X	X	X	X	X	X	X
TPFxd_ERROR_SDAunavail (-15 )															
TPFxd_ERROR_SDAwarning (-16)															
TPFxd_ERROR_VSNinuse (-17)		X	X												
TPFxd_ERROR_noToken (-18)		X	X	X	X			X	X	X	X	X	X	X	X
TPFxd_ERROR_unableToPosition (-19)		X	X	X	X			X							
TPFxd_ERROR_writeOnly (-20)		X	X								X		X		
TPFxd_ERROR_EOVerror (-21)			X							X	X	X	X		
TPFxd_ERROR_alreadyOpen (-22)		X													
TPFxd_ERROR_noBlock (-23)												X			
TPFxd_ERROR_notBaseSSU (-24)	X	X	X												
TPFxd_ERROR_noArchiveGroup (-25)	X	X	X												
TPFxd_ERROR_noArchiveSDA (-26)		X	X												
TPFxd_ERROR_libraryNotFound (-27)		X	X												
TPFxd_ERROR_tapeBlocked (-28)		X	X								X		X		
TPFxd_ERROR_logic (-29)		X	X								X		X		

---

## TPFxd\_archiveEnd—Inform the Archive Facility That the Request Has Ended

This function informs the archive facility that the request has ended and the external device is no longer needed.

### Format

```
#include <c$tpxd.h>
long TPFxd_archiveEnd (TPFxd_extToken *token);
```

#### **token**

The returned token from the TPFxd\_archiveStart request.

### Normal Return

A return code of 1 indicates a normal return.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

## Programming Considerations

- If the TPFxd\_open request was completed successfully, a TPFxd\_close function must be completed before issuing a TPFxd\_archiveEnd request.
- The token that was allocated when the TPFxd\_archiveStart function was issued will be released.
- The TPFxd\_archiveEnd request will ensure that any data written to the external device has been written successfully to the device and is not residing in any host or device buffers.

## Examples

The following example starts the archive process for a device that allows immediate data retrieval and then ends the archive process.

```
#include <c$tpxd.h>
long example()
{
    TPFxd_extToken    *token;
    enum              tpxd_mode mode;
    enum              tpxd_opts access;

    token = NULL;
    mode = WT;
    access = IMMEDIATE;
    TPFxd_archiveStart (&token, mode, access);

    :

    TPFxd_archiveEnd (token);
    :
}
```

## Related Information

- “TPFxd\_archiveStart—Start the Archive Support Facility” on page 1361
- “TPFxd\_close—Signal the End of a TPFxd\_open Request” on page 1363
- “TPFxd\_open—Fulfill Any Mount or Positioning Required” on page 1374.

## TPFxd\_archiveStart—Start the Archive Support Facility

This function opens the archive interface and activates archiving external device support.

### Format

```
#include <c$tpxd.h>
long TPFxd_archiveStart (    TPFxd_extToken **token,
                           enum tpxd_mode    mode,
                           enum tpxd_opts    options);
```

#### token

A pointer to the token that will be allocated for this request.

#### mode

The mode is one of the following:

##### WT

Open for a nonshared write.

##### WS

Open for a shared write.

#### options

The options are listed in the c\$tpxd.h header file.

### Normal Return

The normal return is a positive value.

The \*\*token will be updated to point to the token that was allocated for this request.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

## Programming Considerations

- Before you attempt a TPFxd\_archiveStart function, ensure that you have defined the archive data definition names (DDNs) to the system.
- Do not use the tape names specified as archive tapes for any other application program.
- TPFxd\_archiveStart requests will not be preserved across an IPL.
- The token that is returned will be used on all subsequent TPFxd\_ requests. The storage allocated for the token will be returned to the system when an TPFxd\_archiveEnd function is called and must not be changed by the application.

## Examples

The following example starts the archive process for a device that allows immediate data retrieval.

```
#include <c$tpxd.h>
long example()

{
    TPFxd_extToken *token;
    enum          tpxd_mode mode;
    enum          tpxd_opts access;
```

## TPFxd\_archiveStart

```
mode = WT;  
access = IMMEDIATE;  
TPFxd_archiveStart (&token, mode, access);  
printf("token address/return code is %i\n",token);  
}
```

## Related Information

- “TPFxd\_archiveEnd—Inform the Archive Facility That the Request Has Ended” on page 1360
- “TPFxd\_close—Signal the End of a TPFxd\_open Request” on page 1363
- “TPFxd\_open—Fulfill Any Mount or Positioning Required” on page 1374.



## TPFxd\_close—Signal the End of a TPFxd\_open Request

This function is called for each TPFxd\_open request to indicate that all the read or write operations are completed successfully and any cleanup needed must be performed in preparation for the next TPFxd\_open or TPFxd\_archiveEnd request.

### Format

```
#include <c$tpxd.h>
long TPFxd_close (TPFxd_extToken  *token)
```

#### token

The returned token from the TPFxd\_archiveStart or TPFxd\_open request.

### Normal Return

A return code of 1 indicates a normal return.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

## Programming Considerations

- The TPFxd\_open function must be called before this request.
- If this is a TPFxd\_close request that follows a TPFxd\_ERROR\_EOVwarning or TPFxd\_ERROR\_EOVeror, the volume serial number (VOLSER) associated with the external device will be labeled and removed. A subsequent TPFxd\_open request will cause a new VOLSER to be obtained. For write requests, the same token will still apply and must be used.

## Examples

The following example shows a TPFxd\_open request after a TPFxd\_archiveStart request; after writing one object to the device, TPFxd\_write will issue a TPFxd\_close request.

```
#include <c$tpxd.h>
long example()
{
    TPFxd_extToken    *token;
    TPFxd_locationMap wherefirst;
    enum              tpxd_mode mode;
    enum              tpxd_opts access;
    long              howbigitis;
    long              howlongtowait;
    char              *message;
    long              returncode;
    char              *stufftowrite;

    howlongtowait = 60;
    howbigitis = 32000;
    message = NULL;
    token = NULL;
    mode = WT;
    access = IMMEDIATE;
    TPFxd_archiveStart (&token, mode, access);
    returncode = TPFxd_open (&token,
                            &wherefirst,
                            howbigitis,
                            howlongtowait,
                            message,
                            mode );
```

## TPFxd\_close

```
stufftowrite = malloc(howbigitis);
TPFxd_write(token,stufftowrite,&howbigitis);
free(stufftowrite);
TPFxd_close (token);
:
}
```

## Related Information

- “TPFxd\_archiveEnd—Inform the Archive Facility That the Request Has Ended” on page 1360
- “TPFxd\_nextVolume—Advance to the Next Volume when Reading or Writing” on page 1372
- “TPFxd\_open—Fulfill Any Mount or Positioning Required” on page 1374
- “TPFxd\_sync—Verify Queued Information” on page 1382.

## TPFxd\_getPosition—Retrieve Current Positioning Information

This function is called to retrieve the current positioning information.

### Format

```
#include <c$tpxd.h>
long TPFxd_getPosition (TPFxd_extToken  *token,
                       TPFxd_location  *positioningString);
```

#### **token**

The returned token from the TPFxd\_archiveStart or TPFxd\_open request.

#### **positioningString**

The location to use to return the current positioning string. This string can then be used to reposition the current position.

### Normal Return

The normal return is a positive value. The positioning string address provided on the function call will have the updated positioning string information in it.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

## Programming Considerations

The TPFxd\_open function must be called before this request.

### Examples

The following example gets the position of the external device before writing an object, writes an object, and positions the device to the point before the object that was just written.

```
#include <c$tpxd.h>
long example()
{
    TPFxd_extToken    *token;
    TPFxd_locationMap wherefirst;
    TPFxd_location    whereIwas;
    enum              tpxd_mode mode;
    enum              tpxd_opts access;
    long              howbigitis;
    long              howlongtwait;
    char              *message;
    long              returncode;
    char              *stufftowrite;

    howlongtwait = 60;
    howbigitis = 32000;
    message = NULL;
    token = NULL;
    mode = WT;
    access = IMMEDIATE;
    TPFxd_archiveStart (&token, mode, access);
    TPFxd_open (&token,
               &wherefirst,
               howbigitis,
               howlongtwait,
               message,
               mode );
    returncode = TPFxd_getPosition (token,
```

## TPFxd\_getPosition

```
                                &whereIwas);
printf("getPosition return code is %i\n",returncode);

stufftowrite = malloc(howbigitis);
TPFxd_write(token,stufftowrite,&howbigitis);
free(stufftowrite);

TPFxd_setPosition (token,
                   &whereIwas);
    :
}
```

## Related Information

- “TPFxd\_getPrevPosition–Retrieve Previous Positioning Information” on page 1367
- “TPFxd\_getVOLSER–Retrieve Current VOLSER and Media Type” on page 1369
- “TPFxd\_getVOLSERlist–Retrieve a List of VOLSERs and Media Type” on page 1371
- “TPFxd\_open–Fulfill Any Mount or Positioning Required” on page 1374
- “TPFxd\_setPosition–Set Current Position of the External Device” on page 1380.

## TPFxd\_getPrevPosition—Retrieve Previous Positioning Information

This function is called to retrieve the position of the record that was just read or written.

### Format

```
#include <c$tpxd.h>
long TPFxd_getPrevPosition (TPFxd_extToken *token,
                           TPFxd_location *positioningString);
```

#### token

The returned token from the TPFxd\_archiveStart or TPFxd\_open request.

#### positioningString

The location to use to return the previous positioning string. This string can then be used to reposition the current position.

### Normal Return

The normal return is a positive value. The positioning string address provided on the function call will have the updated positioning string information in it.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

## Programming Considerations

The TPFxd\_open function must be called before this request.

### Examples

The following example writes an object to an external device and then position the device to the point before the object that was just written.

```
#include <c$tpxd.h>
long example()
{
    TPFxd_extToken    *token;
    TPFxd_locationMap wherefirst;
    TPFxd_location    whereIwas;
    enum              tpxd_mode mode;
    enum              tpxd_opts access;
    long              howbigitis;
    long              howlongtwait;
    char              *message;
    long              returncode;
    char              *stufftowrite;

    howlongtwait = 60;
    howbigitis = 32000;
    message = NULL;
    token = NULL;
    mode = WT;
    access = IMMEDIATE;
    TPFxd_archiveStart (&token, mode, access);
    TPFxd_open (&token,
               &wherefirst,
               howbigitis,
               howlongtwait,
               message,
               mode );
    stufftowrite = malloc(howbigitis);
```

## TPFxd\_getPrevPosition

```
TPFxd_write(token,stufftowrite,&howbigitis);
free(stufftowrite);

returncode = TPFxd_getPrevPosition (token,
                                     &whereIwas);
printf("getPrevPosition return code is %i\n",returncode);
TPFxd_setPosition (token,
                  &whereIwas);
:
}
```

## Related Information

- “TPFxd\_getPosition–Retrieve Current Positioning Information” on page 1365
- “TPFxd\_getVOLSER–Retrieve Current VOLSER and Media Type” on page 1369
- “TPFxd\_getVOLSERlist–Retrieve a List of VOLSERs and Media Type” on page 1371
- “TPFxd\_open–Fulfill Any Mount or Positioning Required” on page 1374
- “TPFxd\_setPosition–Set Current Position of the External Device” on page 1380.

## TPFxd\_getVOLSER—Retrieve Current VOLSER and Media Type

This function is called to copy the volume serial number (VOLSER) and device type from a position string into the specified location.

### Format

```
#include <c$tpxd.h>
long TPFxd_getVolser (TPFxd_location  *positioningString,
                     char             *volser,
                     char             *devType,
```

#### positioningString

The location from which the VOLSER and device type will be derived.

#### volser

A pointer to where the VOLSER will be placed.

#### devType

A pointer to where the device type will be placed.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

### Programming Considerations

TPFxd\_getVOLSER does not require the external device to be open. This function can be called using a **TPFxd\_location** parameter, which was obtained from an earlier TPFxd\_getPosition or TPFxd\_getPrevPosition request.

### Examples

The following example writes an object to an external device and then uses the TPFxd\_getVOLSER function to retrieve the VOLSER that the object was just written to.

```
#include <c$tpxd.h>
long example()
{
    TPFxd_extToken    *token;
    TPFxd_locationMap wherefirst;
    TPFxd_location    whereIwas;
    enum              tpxd_mode mode;
    enum              tpxd_opts access;
    long              howbigitis;
    long              howlongtowait;
    char              *message;
    long              returncode;
    char              *stufftowrite;
    char              currentVolser[6];
    char              deviceType;

    howlongtowait = 60;
    howbigitis = 32000;
    message = NULL;
    token = NULL;
    mode = WT;
    access = IMMEDIATE;
    TPFxd_archiveStart (&token, mode, access);
    TPFxd_open (&token,
```

## TPFxd\_getVOLSER

```
        &wherefirst,
        howbigitis,
        howlongtowait,
        message,
        mode );
stufftowrite = malloc(howbigitis);
TPFxd_write(token,stufftowrite,&howbigitis);
free(stufftowrite);

TPFxd_getPosition (token,
                   &whereIwas);
TPFxd_getVOLSER (&whereIwas,
                 (char *) currentVolser,
                 deviceType);
printf("VOLSER is %6.6s",currentVolser);
    :
}
```

## Related Information

- “TPFxd\_getPosition—Retrieve Current Positioning Information” on page 1365
- “TPFxd\_getPrevPosition—Retrieve Previous Positioning Information” on page 1367
- “TPFxd\_setPosition—Set Current Position of the External Device” on page 1380.



## TPFxd\_getVOLSERlist—Retrieve a List of VOLSERS and Media Type

This function is called to copy the VOLSER list and device type from a position string map into the specified location.

### Format

```
#include <c$tpxd.h>
long TPFxd_getVolserList (TPFxd_locationMap *positioningStringMap,
                        char                volser[66],
                        char                *devType);
```

#### positioningStringMap

The location from which the volume serial number (VOLSER) list and device type will be derived.

#### volser

A pointer to where the VOLSERS will be placed.

#### devType

A pointer to where the device type will be placed.

### Normal Return

The normal return is a positive value.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

### Programming Considerations

TPFxd\_getVOLSERlist does not require the external device to be open. This function can be called using a **TPFxd\_location** parameter, which was obtained from an earlier TPFxd\_open or TPFxd\_nextVolume request.

### Examples

The following example takes the positioning string map that was passed and retrieves the list of VOLSERS that were used.

```
#include <c$tpxd.h>
long example(TPFxd_locationMap *positioningStringMap)
{
    long          returncode;
    char          VolserList[66];
    char          deviceType;

    returncode = TPFxd_getVolserList(positioningStringMap,
                                    volserList,
                                    deviceType);
    printf("VOLSER is %6.6s",currentVolser);
    :
}
```

### Related Information

- “TPFxd\_getVOLSER—Retrieve Current VOLSER and Media Type” on page 1369
- “TPFxd\_nextVolume—Advance to the Next Volume when Reading or Writing” on page 1372
- “TPFxd\_open—Fulfill Any Mount or Positioning Required” on page 1374.

## TPFxd\_nextVolume—Advance to the Next Volume when Reading or Writing

This function causes external device support to advance to the next volume when reading or writing and when the TPFxd\_open request has spanned several volumes.

### Format

```
#include <c$tpxd.h>
long TPFxd_nextVolume (TPFxd_extToken    **token,
                      TPFxd_locationMap *positioningStringMap,
                      long               timeout,
                      char               *message);
```

#### token

This pointer must point to the token from a previous TPFxd\_archiveStart function.

#### positioningStringMap

A pointer to a positioning string map. This must be the same positioning string map that was used when the TPFxd\_open request, was issued to position the external device to the correct location. If this is an TPFxd\_open write request, the positioning string, on return, will contain the current position of the external device associated with the token.

#### timeout

The amount of time in seconds to wait before determining the next volume is not available and an error will be returned. If a value of zero is provided, the function will wait indefinitely for the request to be completed.

#### message

A pointer to a message string that will be sent to the console if the function is canceled because of a timeout condition. If a NULL pointer is passed, no message will be sent.

### Normal Return

The normal return is a positive value. For a TPFxd\_nextVolume request following a TPFxd\_open request for a read, the next volume listed in the positioning string will be mounted and positioned as indicated by the **positioningStringMap** parameter. For a TPFxd\_nextVolume request following a TPFxd\_open for write request, a new volume will be allocated and the **positioningStringMap** parameter will be updated to indicate that this volume is now part of the set associated with this TPFxd\_open request.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

### Programming Considerations

This function can be called at any time an external device is opened. The following restrictions apply:

- If the TPFxd\_nextVolume function is entered for an external device that was opened for a read or read-shared request and there are no more volumes in the **TPFxd\_locationMap** parameter that is passed to the function, a TPFxd\_ERROR\_EOError error will be returned.
- If TPFxd\_nextVolume function is issued for an external device that was opened for write or write-shared requests and there are already archive\_max\_volumes (as

defined in the c\$tpxd.h header file) volumes in the **TPFxd\_locationMap** parameter that were passed to the function, a TPFxd\_ERROR\_EOVerror error will be returned.

## Examples

The following example issues a TPFxd\_open function after a TPFxd\_archiveStart function and if a TPFxd\_ERROR\_EOVwarning error occurs when attempting to write the object to the device, a request will be sent for a subsequent volume to be allocated.

```
#include <c$tpxd.h>
long example()
{
    TPFxd_extToken    *token;
    TPFxd_locationMap wherefirst;
    enum              tpxd_mode mode;
    enum              tpxd_opts access;
    long              howbigitis;
    long              howlongtwait;
    char              *message;
    long              returncode;
    char              *stufftowrite;

    howlongtwait = 60;
    howbigitis = 32000;
    message = NULL;
    token = NULL;
    mode = WT;
    access = IMMEDIATE;
    TPFxd_archiveStart (&token, mode, access);
    returncode = TPFxd_open (&token,
                            &wherefirst,
                            howbigitis,
                            howlongtwait,
                            message,
                            mode );

    stufftowrite = malloc(howbigitis);
    returncode = TPFxd_write(token,stufftowrite,&howbigitis);
    if (returncode == TPFxd_ERROR_EOVwarning)
    {
        TPFxd_nextVolume (&token,
                          &wherefirst,
                          howlongtwait,
                          "Why did it timeout?");
    }
    free(stufftowrite);
    :
}
```

## Related Information

- “TPFxd\_close—Signal the End of a TPFxd\_open Request” on page 1363
- “TPFxd\_getVOLSERlist—Retrieve a List of VOLSERs and Media Type” on page 1371
- “TPFxd\_open—Fulfill Any Mount or Positioning Required” on page 1374.

## TPFxd\_open–Fulfill Any Mount or Positioning Required

This function causes external device support to fulfill any mount or positioning required based on the positioning string provided.

### Format

```
#include <c$tpxd.h>
long TPFxd_open (TPFxd_extToken **token,
                 TPFxd_locationMap *positioningStringMap,
                 long size,
                 long timeout,
                 char *message,
                 enum tpxd_mode mode);
```

#### token

This pointer must point to the token from a previous TPFxd\_archiveStart function for write requests.

#### positioningStringMap

A pointer to a positioning string map. If this is a TPFxd\_open read request, the positioning string map will be used to position the external device to the correct location. If this is a TPFxd\_open write request, on return the positioning string map will contain the current position of the external device associated with the token.

#### size

The approximate size of the collection to be written in 4-KB blocks. This will be ignored for read requests.

#### timeout

The amount of time in seconds to wait before determining that the tape is not available and an error will be returned. If a zero value is provided, the function will wait indefinitely for the request to be completed successfully.

#### message

A pointer to a message string that will be sent to the console if the function is canceled because of a timeout condition. If a NULL pointer is passed, no message will be sent.

#### mode

The mode is one of the following:

##### RD

Open for a nonshared read.

##### RS

Open for a shared read.

##### WT

Open for a nonshared write.

##### WS

Open for a shared write.

### Normal Return

The normal return is a positive value. For a TPFxd\_open read request, the token parameter will be updated to the token allocated for this request.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

## Programming Considerations

- Any TPFxd\_open request that is successful must be paired with a TPFxd\_close request before exiting the entry control block (ECB) or a CTL-0013 system error will occur.
- On return from a successful TPFxd\_open function call, the external device is assigned to that ECB and no other ECB can read from or write to it.
- The token that is returned will be used on all subsequent TPFxd\_ requests. The storage allocated for the token will be returned to the system when a TPFxd\_archiveEnd function is called and must not be changed by the application.

## Examples

The following example issues a TPFxd\_open after a TPFxd\_archiveStart request. The estimated size of the object to be written is 32 000 bytes and, if the device is not available in 60 seconds, a TPFXD\_ERROR\_timeout error will be reported. There will also be no error message issued from the TPFxd\_open function when this happens.

```
#include <c$tpxd.h>
long example()
{
    TPFxd_extToken    *token;
    TPFxd_locationMap wherefirst;
    enum              tpxd_mode mode;
    enum              tpxd_opts access;
    long              howbigitis;
    long              howlongtwait;
    char              *message;
    long              returncode;

    howlongtwait = 60;
    howbigitis = 32000;
    message = NULL;
    token = NULL;
    mode = WT;
    access = IMMEDIATE;
    token = TPFxd_archiveStart (mode, access);
    returncode = TPFxd_open (&token,
                            &wherefirst,
                            howbigitis,
                            howlongtwait,
                            message,
                            mode );
    printf("open return code is %i\n",returncode);
    :
}
```

## Related Information

- “TPFxd\_archiveEnd—Inform the Archive Facility That the Request Has Ended” on page 1360
- “TPFxd\_archiveStart—Start the Archive Support Facility” on page 1361
- “TPFxd\_close—Signal the End of a TPFxd\_open Request” on page 1363
- “TPFxd\_getVOLSER—Retrieve Current VOLSER and Media Type” on page 1369
- “TPFxd\_getVOLSERlist—Retrieve a List of VOLSERs and Media Type” on page 1371
- “TPFxd\_nextVolume—Advance to the Next Volume when Reading or Writing” on page 1372.

## TPFxd\_read—Read from an External Device into a Buffer

This function is used to read data from an external device into a buffer in the malloc area. The maximum read size is 32 000 bytes.

### Format

```
#include <c$tpxd.h>
long TPFxd_read (TPFxd_extToken *token,
                 u_char          *buffer,
                 long             *bufferLength);
```

#### token

The returned token from the TPFxd\_archiveStart or TPFxd\_open request.

#### buffer

The pointer to the start of the buffer in which to read the data.

#### bufferLength

The pointer to the length of the buffer that the data will be read into. The buffer should be large enough to hold the data to be returned plus 16 additional bytes for a header.

### Normal Return

The normal return is a positive value; it is the number of bytes read.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

### Programming Considerations

- The TPFxd\_open function must be called before this request.
- If the size of the record is larger than the length provided in the TPFxd\_read function call, only the portion of the record that fits in the area will be returned. The remaining data will be discarded.
- If a TPFxd\_ERROR\_EOError return code is received, there is no data returned and any subsequent TPFxd\_read function calls will continue to receive TPFxd\_ERROR\_EOError return code.
- The buffer length provided on the function call will be adjusted to reflect the actual number of bytes read if the record is smaller than the size of the buffer.

### Examples

The following example reads the first record from the position indicated.

```
#include <c$tpxd.h>
long example(TPFxd_locationMap wherefirst)
{
    TPFxd_extToken    *token;
    enum              tpxd_mode mode;
    long              howbigitis;
    long              howlongtowait;
    char              *message;
    char              *whatWasRead;
    long              returncode;

    howlongtowait = 60;
    howbigitis = 32000;
    message = NULL;
    token = NULL;
```

```

mode = RD;
TPFxd_open (&token,
            &wherefirst,
            howbigitis,
            howlongtowait,
            message,
            mode );
malloc (whatWasRead,howbigitis);
returncode = TPFxd_read (token,
                        whatWasRead);
    :
}

```

## Related Information

- “TPFxd\_close—Signal the End of a TPFxd\_open Request” on page 1363
- “TPFxd\_nextVolume—Advance to the Next Volume when Reading or Writing” on page 1372
- “TPFxd\_open—Fulfill Any Mount or Positioning Required” on page 1374
- “TPFxd\_readBlock—Read into a Core Block from an External Device” on page 1378
- “TPFxd\_setPosition—Set Current Position of the External Device” on page 1380
- “TPFxd\_write—Write to an External Device” on page 1384
- “TPFxd\_writeBlock—Write Core Block Images to the External Device” on page 1386.

## TPFxd\_readBlock—Read into a Core Block from an External Device

This function is used to read core block images from the external device.

### Format

```
#include <c$tpxd.h>
long TPFxd_readBlock (TPFxd_extToken *token,
                     enum t_lvl      level,
                     enum t_blktype   coreBlockType);
```

#### token

The returned token from the TPFxd\_archiveStart or TPFxd\_open request.

#### level

One of 16 possible values representing a valid data level from enumeration type `t_lvl`, expressed as Dx, where x represents the hexadecimal number of the level (0–F). This parameter identifies the data level where the block will be placed.

#### coreBlockType

The core block type to use for the read.

### Normal Return

A return code of 1 indicates a normal return.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

### Programming Considerations

- The TPFxd\_open function must be called before this request.
- If the size of the record is larger than the length implied by the core block type, only the portion of the record that fits in the area will be returned. The remaining data will be discarded.
- If a TPFxd\_ERROR\_EOError return code is received, there is no data returned and any subsequent TPFxd\_readBlock function calls will continue to receive TPFxd\_ERROR\_EOError return code.

### Examples

The following example reads the first record from the position indicated into data level DF.

```
#include <c$tpxd.h>
long example(TPFxd_locationMap wherefirst)
{
    TPFxd_extToken    *token;
    enum              tpxd_mode mode;
    long              howbigitis;
    long              howlongtowait;
    char              *message;
    long              returncode;

    howlongtowait = 60;
    howbigitis = 32000;
    message = NULL;
    token = NULL;
    mode = RD;
    TPFxd_open (&token,
               &wherefirst,
```



```

        howbigitis,
        howlongtowait,
        message,
        mode );
malloc (whatWasRead,howbigitis);
returncode = TPFxd_readBlock(token,DF,L4);
printf("readBlock return code is %i\n",returncode);
:
}

```

## Related Information

- “TPFxd\_close—Signal the End of a TPFxd\_open Request” on page 1363
- “TPFxd\_nextVolume—Advance to the Next Volume when Reading or Writing” on page 1372
- “TPFxd\_open—Fulfill Any Mount or Positioning Required” on page 1374
- “TPFxd\_read—Read from an External Device into a Buffer” on page 1376
- “TPFxd\_setPosition—Set Current Position of the External Device” on page 1380
- “TPFxd\_write—Write to an External Device” on page 1384
- “TPFxd\_writeBlock—Write Core Block Images to the External Device” on page 1386.

## TPFxd\_setPosition–Set Current Position of the External Device

This function is called to position the external device to the requested location based on the information returned from a previous TPFxd\_open or TPFxd\_getPosition request.

### Format

```
#include <c$tpxd.h>
long TPFxd_setPosition (TPFxd_extToken *token,
                       TPFxd_location *positioningString)
```

#### token

The returned token from the TPFxd\_archiveStart or TPFxd\_open request.

#### positioningString

The returned positioning string from a TPFxd\_getPosition function call or the following keyword string:

##### First

The position to the first data record on the external device.

### Normal Return

A return code of 1 indicates a normal return.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

## Programming Considerations

- The TPFxd\_open function must be called before this request.
- This function cannot be used to change from one external volume serial number (VOLSER) to another. To change VOLSERs, issue a TPFxd\_close request followed by another TPFxd\_open request.
- If the external device is opened for output, this function cannot be used to set the position to beyond the current position.

## Examples

The following example issues a TPFxd\_archiveStart and then a TPFxd\_open request, and later repositions the device to the point of the TPFxd\_open request.

```
#include <c$tpxd.h>

long example()
{
    TPFxd_extToken    *token;
    TPFxd_locationMap wherfirst;
    TPFxd_location    wherenow;
    enum              tpxd_mode mode;
    long               howbigitis;
    long               howlongtowait;
    char               *message;
    long               returncode;

    howlongtowait = 60;
    howbigitis = 32000;
    message = NULL;
    mode = WT;
    TPFxd_archiveStart (&token, mode, access);
```

```
returncode = TPFxd_open (&token,  
                        &wherefirst,  
                        howbigitis,  
                        howlongtowait,  
                        message,  
                        mode );  
  
:  
returncode = TPFxd_getPosition (token,  
                                &wherenow);  
:  
  
returncode = TPFxd_setPosition (token,  
                                &wherenow);  
printf("setPosition complete with return code %i\n",returncode);  
}
```

## Related Information

- “TPFxd\_getPosition–Retrieve Current Positioning Information” on page 1365
- “TPFxd\_getPrevPosition–Retrieve Previous Positioning Information” on page 1367.

## TPFxd\_sync–Verify Queued Information

This function is used to ensure that any information queued to the external device has successfully been written to the device. Before a TPFxd\_sync request, any data may still reside in a host or device buffer.

### Format

```
#include <c$tpxd.h>
long TPFxd_sync (TPFxd_extToken *token);
```

#### token

The returned token from the TPFxd\_archiveStart or TPFxd\_open request.

### Normal Return

A return code of 1 indicates a normal return.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

## Programming Considerations

- The TPFxd\_open function must be called before this request.
- If the TPFxd\_sync function returns a TPFxd\_ERROR\_EOVwarning return code, the request will cause the tape to be closed and removed. A subsequent TPFxd\_open request will result in a new tape being mounted and, consequently, the volume serial number (VOLSER) would be changed.

## Examples

The following example writes an object to an external device and then issues a TPFxd\_sync request to ensure that the object was successfully written to the device and does not reside in a hardware or software buffer.

```
#include <c$tpxd.h>
long example()
{
    TPFxd_extToken    *token;
    TPFxd_locationMap wherefirst;
    TPFxd_location    whereIwas;
    enum              tpxd_mode mode;
    enum              tpxd_opts access;
    long              howbigitis;
    long              howlongtowait;
    char              *message;
    long              returncode;
    char              *stufftowrite;

    howlongtowait = 60;
    howbigitis = 32000;
    message = NULL;
    token = NULL;
    mode = WT;
    access = IMMEDIATE;
    TPFxd_archiveStart (&token, mode, access);
    TPFxd_open (&token,
                &wherefirst,
                howbigitis,
                howlongtowait,
                message,
                mode );
```

```
stufftowrite = malloc(howbigitis);
TPFxd_write(token,stufftowrite,&howbigitis);

returncode = TPFxd_sync (token);
printf("sync complete with return code %i\n",returncode);

free(stufftowrite);
:
}
```

## Related Information

- “TPFxd\_close—Signal the End of a TPFxd\_open Request” on page 1363
- “TPFxd\_open—Fulfill Any Mount or Positioning Required” on page 1374.

## TPFxd\_write—Write to an External Device

This function is used to write data from the malloc area to an external device. The maximum write size is 32 000 bytes.

### Format

```
#include <c$tpxd.h>
long TPFxd_write (TPFxd_extToken *token,
                  u_char          *data,
                  long            *dataLength);
```

#### token

The returned token from the TPFxd\_archiveStart or TPFxd\_open request.

#### data

The pointer to the start of the data to write.

#### dataLength

The pointer to the length of the data to write.

### Normal Return

A return code of 1 indicates a normal return.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

### Programming Considerations

- The TPFxd\_open function must be called before this request.
- If the TPFxd\_write function returns a TPFxd\_ERROR\_EOVwarning return code, a TPFxd\_close request will cause the tape to be closed and removed. A subsequent TPFxd\_open request would result in a new tape being mounted and, consequently, the volume serial number (VOLSER) would be changed.
- If a TPFxd\_ERROR\_EOVwarning return code is received, the record has been successfully written but the amount of space remaining on that volume is limited. Additional TPFxd\_write requests will be allowed and will receive a TPFxd\_ERROR\_EOVwarning return code.

### Examples

The following example writes a core area to an external device.

```
#include <c$tpxd.h>
long example()
{
    TPFxd_extToken    *token;
    TPFxd_locationMap wherefirst;
    enum              tpxd_mode mode;
    enum              tpxd_opts access;
    long              howbigitis;
    long              howlongtowait;
    char              *message;
    long              returncode;
    char              *stufftowrite;

    howlongtowait = 60;
    howbigitis = 32000;
    message = NULL;
    token = NULL;
    mode = WT;
```

```

access = IMMEDIATE;
TPFxd_archiveStart (&token, mode, access);
TPFxd_open (&token,
            &wherefirst,
            howbigitis,
            howlongtowait,
            message,
            mode );
stufftowrite = malloc(howbigitis);
:
returncode = TPFxd_write(token,stufftowrite,&howbigitis);
printf("write complete with return code %i\n",returncode);
free(stufftowrite);
:
}

```

## Related Information

- “TPFxd\_close—Signal the End of a TPFxd\_open Request” on page 1363
- “TPFxd\_getPosition—Retrieve Current Positioning Information” on page 1365
- “TPFxd\_getPrevPosition—Retrieve Previous Positioning Information” on page 1367
- “TPFxd\_nextVolume—Advance to the Next Volume when Reading or Writing” on page 1372
- “TPFxd\_open—Fulfill Any Mount or Positioning Required” on page 1374
- “TPFxd\_read—Read from an External Device into a Buffer” on page 1376
- “TPFxd\_readBlock—Read into a Core Block from an External Device” on page 1378
- “TPFxd\_sync—Verify Queued Information” on page 1382
- “TPFxd\_writeBlock—Write Core Block Images to the External Device” on page 1386.

## TPFxd\_writeBlock—Write Core Block Images to the External Device

This function is used to write core block images to the external device.

### Format

```
#include <c$tpxd.h>
long TPFxd_writeBlock (    TPFxd_extToken  *token,
                          enum t_lvl       level);
```

#### token

The returned token from the TPFxd\_archiveStart or TPFxd\_open request.

#### level

One of 16 possible values representing a valid data level from the enumeration type t\_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). This parameter identifies the data level where the block will be placed.

### Normal Return

A return code of 1 indicates a normal return.

### Error Return

An error return is indicated by a negative return code. For a list of error codes applicable to this function, see “Error Codes” on page 1358.

### Programming Considerations

- The TPFxd\_open function must be called before this request.
- If a TPFxd\_writeBlock function returns a TPFxd\_ERROR\_EOVwarning or TPFxd\_ERROR\_EOVerror return code, the next TPFxd\_close request will cause the tape to be closed and removed. A subsequent TPFxd\_open request will result in a new tape being mounted and, consequently, the volume serial number (VOLSER) will be changed.
- If a TPFxd\_ERROR\_EOVwarning return code is received, the record has been successfully written but the amount of space remaining on that volume is limited. Additional TPFxd\_writeBlock requests will be allowed and will receive a TPFxd\_ERROR\_EOVwarning return code.

### Examples

The following example writes a core block to an external device.

```
#include <c$tpxd.h>
long example()
{
    TPFxd_extToken  *token;
    TPFxd_locationMap wherefirst;
    enum            tpxd_mode mode;
    enum            tpxd_opts access;
    long            howbigitis;
    long            howlongtowait;
    char            *message;
    long            returncode;
    char            *stufftowrite;

    howlongtowait = 60;
    howbigitis = 32000;
    message = NULL;
    token = NULL;
    mode = WT;
```



```

access = IMMEDIATE;
TPFxd_archiveStart (&token, mode, access);
TPFxd_open (&token,
            &wherefirst,
            howbigitis,
            howlongtowait,
            message,
            mode );
getcc(DF, GETCC_TYPE+GETCC_FILL, L4, 'A');
returncode = TPFxd_writeBlock(token,DF);
printf("write complete with return code %i\n",returncode);
:
}

```

## Related Information

- “TPFxd\_close—Signal the End of a TPFxd\_open Request” on page 1363
- “TPFxd\_getPosition—Retrieve Current Positioning Information” on page 1365
- “TPFxd\_getPrevPosition—Retrieve Previous Positioning Information” on page 1367
- “TPFxd\_nextVolume—Advance to the Next Volume when Reading or Writing” on page 1372
- “TPFxd\_open—Fulfill Any Mount or Positioning Required” on page 1374
- “TPFxd\_read—Read from an External Device into a Buffer” on page 1376
- “TPFxd\_readBlock—Read into a Core Block from an External Device” on page 1378
- “TPFxd\_sync—Verify Queued Information” on page 1382
- “TPFxd\_write—Write to an External Device” on page 1384.

**TPFxd\_writeBlock**

---

## Appendix A. Conformance of TPF C Support to ANSI/ISO Standards

TPF C language support is a hosted implementation that conforms to ANSI and ISO standard C as defined in ANSI/ISO 9899-1990. A hosted implementation supports:

- The complete C language specification
- The `main` function invoked at program startup
- All of the functions, macros, type definitions, and objects defined in the standard C library.

ANSI/ISO standard C permits some latitude in each of these areas and requires from a conforming implementation a document describing specifically how they are handled.

**Note:** TARGET(TPF) C does not conform to ANSI/ISO standard C.

In the case of TPF C language support:

- Conformance to the C language specification depends on the compiler. All C compilers supported by the TPF system are ANSI/ISO standard C compliant and their specific implementation of the standard is described in an appendix to the compiler *Language Reference*.
- Support for the `main` function and its arguments is described in *TPF Application Programming*.
- The remainder of this chapter describes the TPF system implementation of the standard C library.

---

## Implementation-Defined Behavior of the TPF C Run-Time Library

### Streams and Files

TPF has a hierarchical file system modelled on (but not completely conforming to) POSIX.1. File names are specified as either:

- Absolute path names, which begin with a leading `/` (slash, 0x61), and traverse a directory path starting from the root directory (`/`); for example: `/etc/bin/ls`.
- Relative path names, which do not have a leading `/`, but begin with a file, subdirectory, or symbolic link name that is contained in the current working directory (`cwd`); for example: `my.subdir/my.file`.

Path name components (names of files, subdirectories, symbolic links) can be up to 256 characters long and can consist of any EBCDIC characters other than `/` (slash, '\x61'), or NUL ('\0'), although for portability it is recommended that path name components contain only the portable file name character set, which consists of the following:

- Letters  
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  
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
- Digits  
0 1 2 3 4 5 6 7 8 9
- Punctuation characters  
. (period, '\x4B') \_ (underscore, '\x6D') - (hyphen, '\x60')

Path names are C (NUL-terminated) strings consisting of path name components separated by slashes. Path names can be up to 1024 characters long.

From a standard C library perspective, the following are implementation-defined aspects of TPF files and streams:

Does the last line of a text stream require a terminating new-line character?

- No.

Do space characters that are written out to a text stream immediately before a new-line character appear when read?

- Yes.

What is the number of NUL characters that can be appended to the end of a binary stream:

- There is no limit.

Where is the file position indicator of an append-mode stream initially positioned?

- At the end of the file. Special files may have no inherent length; the position indicator has no meaning for such files.

Does writing on a text stream cause the associated file to be truncated?

- No, the written characters overlay any preexisting characters in the file.

**Note:** Opening a file for *write* access, for example `fopen("my.file", "w");`, truncates the file to 0 bytes when it is opened, so that any previous contents are lost.

Can a file have a length of zero?

- Yes.

Can the same file be simultaneously opened multiple times?

- Yes, either by the same process or by multiple processes.

The effect of the `remove()` function on an open file:

- The link to the file is removed from its parent directory, but the file remains accessible to any process that has it open, until all of the open instances of the file are closed, or until the i-node of the file is allocated again.

The effect of the `rename` function on a file name that exists before to the function call:

- The previous link is removed and replaced by the new link. If no links to the old file remain, the file is deleted.

Are temporary files (created by the `tmpnam` function) removed if the program ends abnormally?

- Temporary files are removed under all circumstances.

**Note:** Files created using a name returned by the `tmpnam` function are not treated specially and must explicitly be removed.

The effect of calling the `tmpnam` function more than `TMP_MAX` times:

- The `tmpnam` function continues to produce names unless it cannot create a name that does not exist as a file, when it returns a null pointer.

The output by the `fprintf` function of a `%p` conversion:

- It is equivalent to `%X`.

The input by the `fscanf` function of a `%p` conversion:

- It is treated as an integer.

The interpretation of a - (dash) character that is neither the first nor the last in the scan list for the `%[ ]` conversion by the `fscanf` function:

- The sequence of characters on either side of the - (dash) are used as delimiters. For example, `%[a-f]` will read characters between 'a' and 'f'.

The value of `errno` on failure of the `fgetpos` and `ftell` functions:

- It depends on the failure. See “`fgetpos`—Get File Position” on page 148 and “`ftell`—Get Current File Position” on page 234 for more information about the `fgetpos` and `ftell` functions.

How does the `abort` function behave with regard to open and temporary files?

- Open files are closed. Temporary files are removed.

## Memory Management

The behavior of the `calloc`, `malloc`, and `realloc` functions if the size requested is zero:

- The `calloc` and `malloc` functions have no effect. The `realloc()` function frees the storage.

## Math

What values do mathematical functions return on domain errors?

- The `acos()`, `asin()`, `atan2()`, `pow()`, and `sqrt()` functions return 0.
- The `log()` and `log10()` functions return `-DBL_MAX`

Do mathematics functions set `errno` to `ERANGE` on underflow range errors?

- Yes.

What is the result of calling the `fmod` function with the second argument zero?

- The `fmod` function returns 0 and does not set `errno`.

## Signals

TPF does not support asynchronous, or hardware signals. See “`signal`—Install Signal Handler” on page 495 and “`raise`—Raise Condition” on page 406 for information about the types of signals supported, default actions, and other aspects of signal handling.

## Error Handling

What message strings does `perror` print and `strerror` return?

- The error message strings are defined in segment `CBSTER`, which is part of the C library load module `CISO`.

## Miscellaneous Functions and Macros

The definition of the NULL macro:

- `#define NULL ((void *)0)`

The format of the diagnostic printed by the assert macro, and its termination behavior:

- When the assert function is executed and the expression is false, it prints a diagnostic with the format:

"Assertion failed: *expression*, file:*filename*, line:*linenumber*\n"

and terminates by calling the abort function. See “assert—Verify Condition or Print Diagnostic Message” on page 15 and “abort—Terminate Program Abnormally” on page 7 for more information about the assert() macro.

The sets of characters tested for by the isalnum(), isalpha(), iscntrl(), islower(), isprint(), and isupper() functions:

- Can vary with the locale. See *TPF Application Programming* for more information about TPF support of C locales.

What status does the exit function return if the value of its argument is other than zero, EXIT\_SUCCESS or EXIT\_FAILURE?

- The value of its argument is returned as exit status.

What are the environment names, and how can the environment list used by the getenv() function be altered?

- There is no restriction on environment names, although the TPF system reserves names beginning with the characters TPF (in any combination of upper and lower case) for its own use.
- A process can alter its environment list using the setenv and unsetenv functions; see “setenv—Add, Change, or Delete an Environment Variable” on page 463 and “unsetenv—Delete an Environment Variable” on page 671 for more information about the setenv and unsetenv functions.
- The common environment list can be altered during restart by loading a version of the UENV real-time user exit program using the TLDR or ALDR loader option; see *TPF System Installation Support Reference* for more information about UENV.

What are the contents of the system function argument, and how is it executed?

- See “system—Execute a Command” on page 527 for more information about the system function.

How are the local time zone and Daylight Saving Time determined?

- These are part of the locale. See *TPF Application Programming* for more information about TPF support of C locales.

What is the era for the clock function?

- The clock function measures the approximate number of microseconds that the ECB has had control of an I-stream since it was created.

---

## Appendix B. Standard C Header Files Supported by the TPF 4.1 System

Following is a list of standard C header files that are supported by the TPF system.

- assert.h
- ctype.h
- errno.h
- float.h
- limits.h
- locale.h
- math.h
- stdarg.h
- stddef.h
- stdio.h
- stdlib.h
- string.h
- sys/ipc.h
- sys/shm.h
- time.h





---

## Appendix C. Compiler Functions Supported as TPF 4.1 Extensions

Following is a list of compiler functions that are supported as TPF system extensions.

- cs
- cds



## Appendix D. C/C++ Functions Supported by the TPF 4.1 System but Not Documented

This section identifies C/C++ functions that are supported by the TPF system but are not documented in the books. Information about these functions is available through other resources, which are identified in each of the following sections.

### Standard C/C++ Library Functions

The following standard C/C++ library functions are supported on the TPF system but not documented in this book; however, they are documented in *OS/390 C/C++ Run-Time Library Reference*.

Table 49. Standard C/C++ Library Functions Supported by the TPF System but Not Documented In This Book

Function	TPF Header File	Standards				C or C++
		ISO-C	POSIX.1	XPG4.1	XPG4.2	
abs	stdlib.h	X	X	X	X	Both
acos	math.h	X	X	X	X	Both
asctime	time.h	X	X	X	X	Both
asin	math.h	X	X	X	X	Both
atan	math.h	X	X	X	X	Both
atan2	math.h	X	X	X	X	Both
atof	stdlib.h	X	X	X	X	Both
atoi	stdlib.h	X	X	X	X	Both
atol	stdlib.h	X	X	X	X	Both
bcmp	strings.h				X	Both
bcopy	strings.h				X	Both
bsearch	stdlib.h	X	X	X	X	Both
bzero	strings.h				X	Both
calloc	stdlib.h	X	X	X	X	Both
ceil	math.h	X	X	X	X	Both
clock	time.h	X	X	X	X	Both
cos	math.h	X	X	X	X	Both
cosh	math.h	X	X	X	X	Both
ctime	time.h	X	X	X	X	Both
difftime	time.h	X		X	X	Both
div	stdlib.h	X		X	X	Both
exp	math.h	X	X	X	X	Both
fabs	math.h	X	X	X	X	Both
ffs	strings.h				X	Both
fgetwc	wchar.h	X		X	X	Both
fgetws	wchar.h	X		X	X	Both
floor	math.h	X	X	X	X	Both
fmod	math.h	X	X	X	X	Both

Table 49. Standard C/C++ Library Functions Supported by the TPF System but Not Documented In This Book (continued)

Function	TPF Header File	Standards				C or C++
		ISO-C	POSIX.1	XPG4.1	XPG4.2	
fnmatch	fnmatch.h			X	X	Both
fputwc	wchar.h			X	X	Both
fputws	wchar.h			X	X	Both
free	stdlib.h	X	X	X	X	Both
frexp	math.h	X	X	X	X	Both
getopt <sup>1</sup>	stdlib.h			X	X	Both
getsyntax	variant.h					Both
getwc	wchar.h	X		X	X	Both
getwchar	wchar.h	X		X	X	Both
getwmccoll	collate.h					Both
glob <sup>2 3</sup>	globb.h			X	X	Both
globfree <sup>2</sup>	globb.h			X	X	Both
gmtime	time.h	X	X	X	X	Both
iconv	iconv.h			X	X	Both
iconv_close	iconv.h			X	X	Both
iconv_open	iconv.h			X	X	Both
isalnum	ctype.h	X	X	X	X	Both
isalpha	ctype.h	X	X	X	X	Both
isblank	ctype.h					Both
iscntrl	ctype.h	X	X	X	X	Both
isdigit	ctype.h	X	X	X	X	Both
isgraph	ctype.h	X	X	X	X	Both
islower	ctype.h	X	X	X	X	Both
isprint	ctype.h	X	X	X	X	Both
ispunct	ctype.h	X	X	X	X	Both
isspace	ctype.h	X	X	X	X	Both
isupper	ctype.h	X	X	X	X	Both
iswalnum	wctype.h	X		X	X	Both
iswalpha	wctype.h	X		X	X	Both
iswcntrl	wctype.h	X		X	X	Both
iswctype	wctype.h	X		X	X	Both
iswdigit	wctype.h	X		X	X	Both
iswgraph	wctype.h	X		X	X	Both
iswlower	wctype.h	X		X	X	Both
iswprint	wctype.h	X		X	X	Both
iswpunct	wctype.h	X		X	X	Both
iswspace	wctype.h	X		X	X	Both
iswupper	wctype.h	X		X	X	Both

Table 49. Standard C/C++ Library Functions Supported by the TPF System but Not Documented In This Book (continued)

Function	TPF Header File	Standards				C or C++
		ISO-C	POSIX.1	XPG4.1	XPG4.2	
iswxdigit	wctype.h	X		X	X	Both
isxdigit	ctype.h	X	X	X	X	Both
labs	stdlib.h	X		X	X	Both
ldexp	math.h	X	X	X	X	Both
ldiv	stdlib.h	X		X	X	Both
llabs	stdlib.h					Both
lldiv	stdlib.h					Both
localeconv	locale.h	X		X	X	Both
localtime	time.h	X	X	X	X	Both
log	math.h	X	X	X	X	Both
log10	math.h	X	X	X	X	Both
malloc	stdlib.h	X	X	X	X	Both
mblen	stdlib.h	X		X	X	Both
mbrlen	wchar.h	X				Both
mbrtowc	wchar.h	X				Both
mbsinit	wchar.h	X				Both
mbsrtowcs	wchar.h	X				Both
mbstowcs	stdlib.h	X		X	X	Both
mbtowc	stdlib.h	X		X	X	Both
memchr	string.h	X		X	X	Both
memcmp	string.h	X		X	X	Both
memcpy	string.h	X		X	X	Both
memmove	string.h	X		X	X	Both
memset	string.h	X		X	X	Both
mktime	time.h	X	X	X	X	Both
modf	math.h	X	X	X	X	Both
nl_langinfo	langinfo.h			X	X	Both
pow	math.h	X	X	X	X	Both
putwc	wchar.h			X	X	Both
putwchar	wchar.h	X		X	X	Both
qsort	stdlib.h	X		X	X	Both
rand	stdlib.h	X	X	X	X	Both
realloc	stdlib.h	X	X	X	X	Both
regcomp	regex.h			X	X	Both
regerror	regex.h			X	X	Both
regexexec	regex.h			X	X	Both
regfree	regex.h			X	X	Both
setlocale	locale.h	X	X	X	X	Both

Table 49. Standard C/C++ Library Functions Supported by the TPF System but Not Documented In This Book (continued)

Function	TPF Header File	Standards				C or C++
		ISO-C	POSIX.1	XPG4.1	XPG4.2	
sin	math.h	X	X	X	X	Both
sinh	math.h	X	X	X	X	Both
sqrt	math.h	X	X	X	X	Both
srand	stdlib.h		X	X	X	Both
strcasecmp	strings.h				X	Both
strcat	string.h	X	X	X	X	Both
strchr	string.h	X		X	X	Both
strcmp	string.h	X	X	X	X	Both
strcoll	string.h	X	X	X	X	Both
strcpy	string.h	X		X	X	Both
strcspn	string.h	X		X	X	Both
strdup	string.h				X	Both
strfmon	monetary.h	X	X	X	X	Both
strftime	time.h	X	X	X	X	Both
strlen	string.h	X	X	X	X	Both
strncasecmp	strings.h				X	Both
strncat	string.h	X	X	X	X	Both
strncmp	string.h	X	X	X	X	Both
strncpy	string.h	X		X	X	Both
strpbrk	string.h	X	X	X	X	Both
strrchr	string.h	X	X	X	X	Both
strspn	string.h	X		X	X	Both
strstr	string.h	X		X	X	Both
strtol	collate.h	X				Both
strtod	stdlib.h	X		X	X	Both
strtok	string.h	X		X	X	Both
strtol	stdlib.h	X		X	X	Both
strtoll	stdlib.h					Both
strtoull	stdlib.h					Both
strtoul	stdlib.h	X		X	X	Both
strxfrm	string.h	X		X	X	Both
swprintf	wchar.h	X				Both
swscanf	wchar.h	X				Both
tan	math.h	X	X	X	X	Both
tanh	math.h	X	X	X	X	Both
time	time.h	X	X	X	X	Both
tolower	ctype.h	X	X	X	X	Both
toupper	ctype.h	X	X	X	X	Both

Table 49. Standard C/C++ Library Functions Supported by the TPF System but Not Documented In This Book (continued)

Function	TPF Header File	Standards				C or C++
		ISO-C	POSIX.1	XPG4.1	XPG4.2	
tolower	wctype.h		X	X	X	Both
toupper	wctype.h		X	X	X	Both
ungetwc	wchar.h			X	X	Both
va_arg	stdarg.h	X				Both
va_end	stdarg.h	X				Both
va_start	stdarg.h	X				Both
vswprintf	wchar.h	X				Both
wcrtomb	wchar.h	X				Both
wcscat	wchar.h	X		X	X	Both
wcschr	wchar.h	X		X	X	Both
wcscmp	wchar.h	X		X	X	Both
wscoll	wchar.h	X		X	X	Both
wcscpy	wchar.h	X		X	X	Both
wcscspn	wchar.h	X		X	X	Both
wcsftime	wchar.h	X		X	X	Both
wcslen	wchar.h	X		X	X	Both
wcsncat	wchar.h	X		X	X	Both
wcsncmp	wchar.h	X		X	X	Both
wcsncpy	wchar.h	X		X	X	Both
wcspbrk	wchar.h	X		X	X	Both
wcsrchr	wchar.h	X		X	X	Both
wcsrtombs	wchar.h	X				Both
wcsspn	wchar.h	X		X	X	Both
wcsstr	wchar.h	X				Both
wctod	wchar.h	X		X	X	Both
wctok	wchar.h	X		X	X	Both
wctol	wchar.h	X		X	X	Both
wctoll	wchar.h					Both
wctombs	stdlib.h	X		X	X	Both
wctoul	wchar.h	X		X	X	Both
wctoull	wchar.h					Both
wcswcs	wchar.h	X		X	X	Both
wcswidth	wchar.h			X	X	Both
wcsxfrm	wchar.h	X		X	X	Both
wctob	wchar.h	X				Both
wctomb	stdlib.h	X		X	X	Both
wctype	wchar.h	X		X	X	Both
wcwidth	wchar.h			X	X	Both

Table 49. Standard C/C++ Library Functions Supported by the TPF System but Not Documented In This Book (continued)

Function	TPF Header File	Standards				C or C++
		ISO-C	POSIX.1	XPG4.1	XPG4.2	
wmemchr	wchar.h	X				Both
wmemcmp	wchar.h	X				Both
wmemcpy	wchar.h	X				Both
wmemmove	wchar.h	X				Both
wmemset	wchar.h	X				Both

**Notes:**

1. The standard header file for getopt is unistd.h.
2. The standard header file for glob and globfree is glob.h.
3. There are two glob functions supported on the TPF 4.1 system. See “glob—Address TPF Global Field or Record” on page 277 for more information on the TPF-specific glob.

## XML Parser for C++ (XML4C) Version 3.5.1 Classes

The following classes are used with the XML4C parser. For information about viewing the documentation for these classes, go to the *XML User's Guide*.

- AttributeList (Interface for an element's attribute specifications)
- Attributes (Interface for an element's attribute specifications)
- Base64
- BinFileInputStream
- BinInputStream
- BinMemInputStream
- ChildNode
- ContentHandler (Receive notification of general document events)
- DefaultHandler (Default base class for SAX2 handlers)
- DocTypeHandler
- DocumentHandler (Receive notification of general document events)
- DOM\_Attr (Refers to an attribute of an XML element)
- DOM\_CDATASection (DOM\_CDATASection objects refer to the data from an XML CDATA section)
- DOM\_CharacterData (The DOM\_CharacterData interface extends Node with a set of methods for accessing character data in the DOM)
- DOM\_Comment (Refers to XML comment nodes in the DOM)
- DOM\_Document (Refers to XML Document nodes in the DOM)
- DOM\_DocumentFragment (A "lightweight" or "minimal" Document object)
- DOM\_DocumentType (Each Document has a doctype whose value is either null or a DocumentType object)
- DOM\_DOMException (Encapsulate a general DOM error or warning)
- DOM\_DOMImplementation (Provides a way to query the capabilities of an implementation of the DOM)
- DOM\_Element (By far, the vast majority of objects (apart from text) that authors encounter when traversing a document are DOM\_Element nodes)



- DOM\_Entity (This interface represents an entity, either parsed or unparsed, in an XML document)
- DOM\_EntityReference (EntityReference nodes will appear in the structure model when an entity reference is in the source document, or when the user wants to insert an entity reference)
- DOM\_NamedNodeMap (NamedNodeMaps are used to represent collections of nodes that can be accessed by name)
- DOM\_Node (The primary data type for the entire Document Object Model)
- DOM\_NodeFilter (Filters are objects that know how to "filter out" nodes)
- DOM\_NodeIterator (NodeIterators are used to step through a set of nodes)
- DOM\_NodeList (The NodeList interface provides the abstraction of an ordered collection of nodes)
- DOM\_Notation (This interface represents a notation declared in the DTD)
- DOM\_ProcessingInstruction (The ProcessingInstruction interface represents a "processing instruction", used in XML as a way to keep processor-specific information in the text of the document)
- DOM\_Range DOM\_RangeException (Encapsulate range-related DOM error or warning)
- DOM\_Text (Represents the textual content, termed character data in XML, of an Element or Attr)
- DOM\_TreeWalker (DOM\_TreeWalker objects are used to navigate a document tree or subtree using the view of the document defined by its whatToShow flags and any filters that are defined for the DOM\_TreeWalker)
- DOM\_XMLDecl (Class to refer to XML Declaration nodes in the DOM)
- DOMParser (This class implements the Document Object Model (DOM) interface)
- DOMString (The generic string class that stores all strings used in the DOM C++ API)
- DTDHandler (Receive notification of basic DTD-related events)
- EntityResolver (Basic interface for resolving entities)
- ErrorHandler (Basic interface for SAX error handlers)
- HandlerBase (Default base class for handlers)
- HexBin
- IDOMParser (This class implements the Document Object Model (DOM) interface)
- InputSource (A single input source for an XML entity)
- LexicalHandler (Receive notification of lexical events)
- LocalFileInputSource (This class is a derivative of the standard InputSource class)
- Locator (Interface for associating a SAX event with a document location)
- MemBufInputSource (This class is a derivative of the standard InputSource class)
- NoDefTranscoderException
- ParentNode (Inherits from ChildImpl and adds the capability of having child nodes)
- Parser (Basic interface for SAX (Simple API for XML) parsers)
- QName
- SAX2XMLReader
- SAXException (Encapsulate a general SAX error or warning)
- SAXNotRecognizedException

- SAXNotSupportedException
- SAXParseException (Encapsulate an XML parse error or warning)
- SAXParser (This class implements the SAX 'Parser' interface and should be used by applications wanting to parse the XML files using SAX)
- StdInInputSource (This class is a derivative of the standard InputSource class)
- XMLTransService::TransRec
- URLInputSource (This class is a derivative of the standard InputSource class)
- XMLAttDef (Represents the core information of an attribute definition)
- XMLAttDefList (This class defines an abstract interface that all validators must support)
- XMLAttr (This class defines the information about an attribute that will come out of the scanner during parsing)
- XMLBigDecimal
- XMLBigInteger
- XMLContentModel (This class defines the abstract interface for all content models)
- XMLDeleter
- XMLDeleterFor
- XMLDocumentHandler (This abstract class provides the interface for the scanner to return XML document information up to the parser as it scans through the document)
- XMLElementDecl (This class defines the core information of an element declaration)
- XMLEntityDecl (This class defines the core information that defines an XML entity, no matter what validator is used)
- XMLEntityHandler (This abstract class is a callback mechanism for the scanner)
- XMLErrorHandler (This abstract class defines a callback mechanism for the scanner)
- XMLErrs
- XMLException
- XMLFormatTarget
- XMLFormatter (This class provides the basic formatting capabilities that are required to turn the Unicode-based XML data from the parsers into a form that can be used on non-Unicode-based systems)
- XMLLCPTranscoder XMLNotationDecl (This class represents the core information about a notation declaration that all validators must at least support)
- XMLPlatformUtils (Utilities that must be implemented in a platform-specific way)
- XMLReaderFactory
- XMLString (Class for representing native character strings and handling common string operations)
- XMLStringTokenizer (The string tokenizer class breaks a string into tokens)
- XMLTranscoder
- XMLTransService
- XMLUni
- XMLValid
- XMLValidator (This abstract class provides the interface for all validators).

---

## Appendix E. Programming Support for the TPF File System

To take advantage of the stream files, directories, and other support of the TPF file system, programmers must use a set of TPF file system C functions.

---

### TPF File System C Functions

TPF file system C functions operate through the TPF stream I/O support. Table 50 lists these functions.

**Note:** All file system application programming interfaces (APIs) are atomic and are completed when control returns to the caller. File system APIs that are called within an application commit scope are not part of that commit scope. A device driver can be written so an update to a special file is within the application commit scope. See “User-Defined Device Driver Functions” on page 1414 for more information.

Table 50. TPF File System C Functions

C function	Description	C Header File	Standards
access	Determine Whether a File Can Be Accessed	unistd.h	POSIX.1 (5.6.3)
atexit	Register Program Termination Function	stdlib.h	ANSI/ISO (7.10.4.2)
chdir	Change the Working Directory	unistd.h	POSIX.1 (5.2.1)
chmod	Change the Mode of a File or Directory	sys/stat.h	POSIX.1 (5.6.4)
chown	Change the Owner or Group of a File or Directory	unistd.h	POSIX.1 (5.6.5)
clearerr	Reset Error and End-of-File	stdio.h	ISO-C (7.9.10.1)
close	Close a File	unistd.h	POSIX.1 (6.3.1)
closedir	Close a Directory	sys/types.h dirent.h	POSIX.1 (5.1.2)
creat	Create a New File or Rewrite an Existing One	sys/types.h sys/stat.h fcntl.h	POSIX.1 (5.3.2)
dup	Duplicate an Open File Descriptor	unistd.h	POSIX.1 (6.2.1)
dup2	Duplicate an Open File Descriptor to Another	unistd.h	POSIX.1 (6.2.1)
fchmod	Change the Mode of a File or Directory by File Descriptor	sys/stat.h	POSIX.1 (5.6.4)
fchown	Change the Owner or Group by File Descriptor	unistd.h	POSIX.1a (5.6.5)
fclose	Close a File	stdio.h	ISO-C (7.9.5.1) POSIX.1 (8.2.3.2)
fcntl	Control Open File Descriptors	sys/types.h unistd.h fcntl.h	POSIX.1 (6.5.2)
fdopen	Associate a Stream with an Open File Descriptor	stdio.h	POSIX.1 (8.2.2)
feof	Test End-of-File Indicator	stdio.h	ISO-C (7.9.10.2)
ferror	Test for Read/Write Errors	stdio.h	ISO-C (7.9.10.3)

Table 50. TPF File System C Functions (continued)

C function	Description	C Header File	Standards
fflush	Write a Buffer to File	stdio.h	ISO-C (7.9.5.2) POSIX.1 (8.2.3.4)
fgetc	Read a Character	stdio.h	ISO-C (7.9.7.1) POSIX.1 (8.2.3.5)
fgetpos	Get File Position	stdio.h	ISO-C (7.9.9.1)
fgets	Read a String from a Stream	stdio.h	ISO-C (7.9.7.2) POSIX.1 (8.2.3.5)
fileno	Get the File Descriptor from an Open Stream	stdio.h	POSIX.1 (8.2.1)
fopen	Open a File	stdio.h	ISO-C (7.9.5.3) POSIX.1 (8.2.3.1)
fprintf	Format and Write Data to a Stream	stdio.h	ISO-C (7.9.6.1) POSIX.1 (8.2.3.6)
fputc	Write a Character	stdio.h	ISO-C (7.9.7.3) POSIX.1 (8.2.3.6)
fputs	Write a String to a Stream	stdio.h	ISO-C (7.9.7.4) POSIX.1 (8.2.3.6)
fread	Read Items	stdio.h	ISO-C (7.9.8.1) POSIX.1 (8.2.3.5)
freopen	Redirect an Open File	stdio.h	ISO-C (7.9.5.4) POSIX.1 (8.2.3.3)
fscanf	Read and Format Data from a Stream	stdio.h	ISO-C (7.9.6.2) POSIX.1 (8.2.3.5)
fseek	Set the File Position	stdio.h	ISO-C (7.9.9.2) POSIX.1 (8.2.3.7)
fsetpos	Set File Position	stdio.h	ISO-C (7.9.9.3)
fstat	Get Status Information about a File	sys/types.h sys/stat.h	POSIX.1 (5.6.2)
fsync	Write Changes to Direct-Access Storage	unistd.h	POSIX.1b (6.6.1)
ftell	Get Current File Position	stdio.h	ISO-C (7.9.9.4) POSIX.1 (8.2.3.10)
ftruncate	Truncate a File	unistd.h	POSIX.1b (5.6.7)
fwrite	Write Items	stdio.h	ISO-C (7.9.8.2) POSIX.1 (8.2.3.6)
getc	Read a Character from a Stream	stdio.h	ISO-C (7.9.7.5) POSIX.1 (8.2.3.5)
getchar	Read a Character from the Standard Input Stream	stdio.h	ISO-C (7.9.7.6) POSIX.1 (8.2.3.5)
getcwd	Get Path Name of the Current Working Directory	unistd.h	POSIX.1 (5.2.2)
link	Create a Link to a File	unistd.h	POSIX.1 (5.3.4)
lseek	Change the Offset of a File	unistd.h sys/types.h	POSIX.1 (6.5.3)
lstat	Get the Status of a File or Symbolic Link	sys/types.h sys/stat.h	POSIX.1a (5.6.2)
mkdir	Make a Directory	sys/stat.h	POSIX.1 (5.4.1)

Table 50. TPF File System C Functions (continued)

C function	Description	C Header File	Standards
mkfifo	Make a FIFO Special File	sys/stat.h sys/types.h	POSIX.1 (5.4.2)
mknod	Make a Special File	sys/stat.h	
open	Open a File	fcntl.h sys/stat.h sys/types.h	POSIX.1 (5.3.1)
opendir	Open a Directory	sys/types.h dirent.h	POSIX.1 (5.1.2)
pipe	Create an Unnamed Pipe	unistd.h	POSIX.1 (6.1.1)
putc	Write a Character to a Stream	stdio.h	ISO-C (7.9.7.8) POSIX.1 (8.2.3.6)
putchar	Write a Character to the Standard Output Stream	stdio.h	ISO-C (7.9.7.9) POSIX.1 (8.2.3.6)
read	Read a File	unistd.h	POSIX.1 (6.4.1)
readdir	Read an Entry from a Directory	sys/types.h dirent.h	POSIX.1 (5.1.2)
readlink	Read the Value of a Symbolic Link	unistd.h	POSIX.1a (5.3.5)
remove	Delete a File	stdio.h	ISO-C (7.9.4.1) POSIX.1 (8.2.4)
rename	Rename a File	stdio.h	ISO-C (7.9.4.2) POSIX.1 (5.5.3)
rewind	Set File Position to Beginning of File	stdio.h	ISO-C (7.9.9.5) POSIX.1 (8.2.3.7)
rewinddir	Reposition a Directory Stream to the Beginning	sys/types.h dirent.h	POSIX.1 (5.1.2)
rmdir	Remove a Directory	unistd.h	POSIX.1 (5.5.2)
setbuf	Control Buffering	stdio.h	ISO-C (7.9.5.5)
setvbuf	Control Buffering Strategy and Buffer Size	stdio.h	ISO-C (7.9.5.6)
sprintf	Format and Write Data to a String	stdio.h	ISO-C (7.9.6.5)
sscanf	Read and Format Data from a Buffer	stdio.h	ISO-C (7.9.6.6)
stat	Get File Information	sys/types.h sys/stat.h	POSIX.1 (5.6.2)
symlink	Create a Symbolic Link to a Path Name	unistd.h	POSIX.1a (5.3.6)
system	Execute a Command	stdlib.h	ISO-C (7.10.4.5) POSIX.1a (8.1.5)
tmpfile	Create a Temporary File	stdio.h	ISO-C (7.9.4.3) POSIX.1 (8.2.3.9)
tmpnam	Produce a Temporary File Name	stdio.h	ISO-C (7.9.4.4)
umask	Set or Display the File Mode Creation Mask	sys/stat.h sys/types.h	POSIX.1 (5.3.3)
ungetc	Push Character onto an Input Stream	stdio.h	ISO-C (7.9.7.11)
unlink	Remove a Directory Entry	unistd.h	POSIX.1 (5.5.1)
utime	Set File Access and Modification Times	utime.h sys/types.h	POSIX.1 (5.6.6)

Table 50. TPF File System C Functions (continued)

C function	Description	C Header File	Standards
vfprintf	Format and Write Data to a Stream	stdarg.h stdio.h	ISO-C (7.9.6.7) POSIX.1 (8.2.3.6)
vprintf	Format and Write Data to the Standard Output Stream	stdarg.h stdio.h	ISO-C (7.9.6.8) POSIX.1 (8.2.3.6)
vsprintf	Format and Write Data to Buffer	stdarg.h stdio.h	ISO-C (7.9.6.9)
write	Write a File	unistd.h	POSIX.1 (6.4.2)
<b>Note:</b> Sockets also supports the close, read, and write C functions.			
<b>Note:</b> References to <b>standards:</b>			
<b>ISO-C</b>	ISO-C Standard, International Standard ISO/IEC 9899:1990 and American National Standard for Programming Languages---C, ANSI/ISO 9899-1990		
<b>POSIX.1</b>	Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language], International Standard ISO/IEC 9945-1:1990 (IEEE Std 1003.1-1990)		
<b>POSIX.1a</b>	P1003.1a—Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]—Amendment, P1003.1a, D12, January 1995, DS1933		
<b>POSIX.2</b>	IEEE Standard for Information Technology---Portable Operating System Interface (POSIX)---Part 1: System Application Program Interface (API)---Amendment 1: Real-time Extension [C language] (IEEE Std 1003.b-1993, formerly known as IEEE P1003.4).		

## Path Name Rules for TPF File System C Functions

When using a TPF file system C function to perform an operation on an object, you identify the object by supplying its path. Following is a summary of rules to remember when specifying path names in a C function. The term *object* in these rules refers to any directory, file, link, or other object.

- Path names are specified in hierarchical order beginning with the highest level of the directory hierarchy. The name of each component in the path is separated by a slash (/); for example:

Dir1/Dir2/Dir3/UsrFile

The back slash (\) is not recognized as a separator. It is handled as just another character in a name.

- Object names must be unique in a directory.
- The maximum length of each component of the path name and the maximum length of the path name string can vary for each file system. For TPF file system C functions, the maximum length of each component of the path name is 256 and the maximum length of the path name string is 1024.
- A / character at the beginning of a path name means that the path begins at the root (/) directory; for example:

/Dir1/Dir2/Dir3/UsrFile

- If the path name does not begin with a / character, the path is assumed to begin at the current directory; for example:

MyDir/MyFile

where MyDir is a subdirectory of the current directory.

- File names are case-sensitive; therefore, Dir1, dir1, and DIR1 are three different directories.
- As in UNIX, directory and file names beginning with a period are, by default, not displayed by the ZFILE ls command. To display these directory and file names, use:

```
ZFILE ls -a
```

## Specifying File Descriptors in System I/O C Functions

System-level I/O functions are used when an application program needs to work on a directory structure or when you need more control over a file than is provided by buffered I/O functions. System-level I/O functions such as `creat`, `open`, `close`, `read`, `write`, `fcntl`, `chdir`, `chown`, `chmod`, `mkdir`, `mknod`, `stat`, and others identify a file by specifying a *file descriptor*.

A file descriptor is a nonnegative integer that is first returned by the `open` or `creat` C function. Each open file has a unique file descriptor. Some TPF file system C functions use the file descriptor to identify an open file when performing operations on the file. File descriptors 0, 1, and 2 are initially reserved for the standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`) streams.

Each file descriptor refers to an *open file description*, which contains information such as a file offset, status of the file, and access modes for the file. The same open file description can be referred to by more than one file descriptor, but a file descriptor can refer to only one open file description.

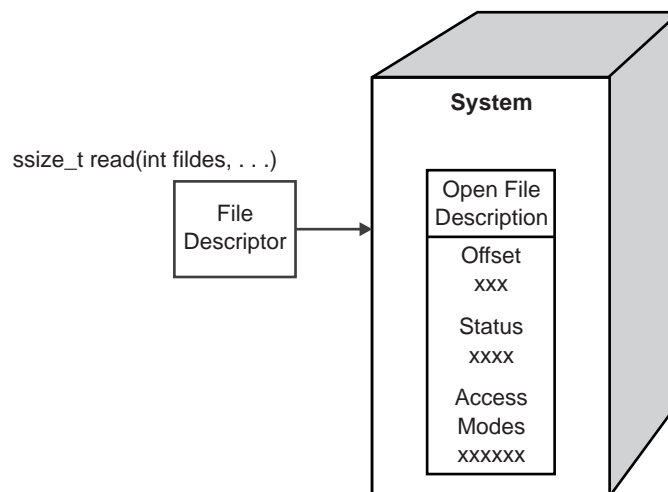


Figure 2. File Descriptor and Open File Description

## Specifying Permissions in TPF File System C Functions

When using TPF file system C functions, you can restrict access to objects by using *permissions*. The types of permissions are read and write (for a file or a directory) or search (for a directory). The permissions are shown by a set of permission bits, which make up the *mode of access* of the file or directory. You can change the permission bits by using the `chmod` or `fchmod` change mode function. You can also use the `umask` function to control which file permission bits are set each time a job creates a file.

---

## TPF-Supplied User and Group Names

Table 51 on page 1411 contains a list of user and group names that are supplied by the TPF system.



Table 51. TPF-Supplied User and Group Names. Users have a primary group, indicated by P, and can be members of other groups, indicated by m. Unless it is explicitly changed (for example, using the *setgid* function) the user's primary group is the effective group used for checking file access modes.

User Name	UID	Group Name	adm	bin	ftp	http	mail	news	nogroup	operator	root	sys	tftp	tpdfiltg	tpfuser
		GID	4	1	25	2	8	13	55	5	0	3	35	257	100
adm	4		P									m			
bin	2			P		m						m			
ftp	19			P	m										
http	3			m		P									
mail	9						m					P			
news	11							m				P			
nobody	65534								P						
operator	13		m							m	P				
postfix	14											P			
postmstr	15						P								
root	0		m	P	m	m	m	m		m	m	m	m	m	m
tftp	22			P									m		
tpdfiltu	256													P	
tpfuser1	260														P
tpfuser2	261														P
tpfuser3	262														P

---

## Adding a User-Defined Device Driver

To add a user-defined device driver to the system:

1. Write the set of device driver functions (described in “User-Defined Device Driver Functions” on page 1414) as required by the interface declared in `c$spif.h`. Define the device driver functions as non-exportable functions in the COMX library.
2. Declare the user-defined device driver functions in a device driver header file. For examples of how to do this, see `c$ddsm.h` (for the sample input and output message drivers) or `i$fsdd.h` (for the IBM-supplied regular and null device drivers). The following is a template for this declaration:

```
/* *****  
/* Declare the functions for the user-defined device driver.      */  
/* *****  
TPF_FSDD_APPEND      new_device_driver_append_function;  
TPF_FSDD_CLOSE       new_device_driver_close_function;  
TPF_FSDD_GET         new_device_driver_get_function;  
TPF_FSDD_OPEN        new_device_driver_open_function;  
TPF_FSDD_PUT         new_device_driver_put_function;  
TPF_FSDD_RESIZE      new_device_driver_resize_function;  
TPF_FSDD_SIZE        new_device_driver_size_function;  
TPF_FSDD_POLL        new_device_driver_poll_function;  
TPF_FSDD_POLL_CLEAN  new_device_driver_poll_clean_function;  
  
/* *****  
/* The following macro defines the initializer for the new struct */  
/* tpf_spif entry in the UDDTBL device driver table.             */  
/* *****  
#define NEW_DEVICE_DRIVER_TABLE_ENTRY {  
    new_device_driver_append_function,  
    new_device_driver_close_function,  
    new_device_driver_get_function,  
    new_device_driver_open_function,  
    new_device_driver_put_function,  
    new_device_driver_resize_function,  
    new_device_driver_size_function,  
    new_device_driver_poll_function,  
    new_device_driver_poll_clean_function  
}
```

3. Using the new or updated header from the previous step, update segment UDDTBL:
  - a. Add the new or updated header from the previous step to the list of `#includes` following the prolog.
  - b. Define a new entry in the device driver interface table by adding the new initializer macro to the initializer list for the `TPF_USRDDTAB` array. The index of the new device driver element in the `TPF_USRDDTAB` array is the major device number for any special file that uses the new device driver.

**Note:** Do not change or reorder any existing entries. If you do, you will have to delete and re-create all special files for which the major device number changed.

4. Recompile UDDTBL.
5. Rebuild the COMX library including the new object for UDDTBL (from step 3) and any user-defined device driver objects (from step 1).
6. Load the rebuilt COMX library online.

7. Enter the ZFILE mknod command to create the character special files that use the user-defined device driver by using the major device number from step 3b on page 1412. See *TPF Operations* for more information about the ZFILE mknod command.

The device driver is now accessible to online load modules by opening the special files defined in step 7.

---

## Special Files

This section describes some considerations for special files.

### Device Driver Flags

In addition to the device driver functions that make up the special file interface, every device driver table entry contains a `tpf_spif_flags` field. The following flags are defined for this field:

#### **TPF\_SPIF\_INHERITABLE**

Allows the file descriptor of the special file to be inherited through the `tpf_fork` function.

#### **TPF\_SPIF\_NODDCLOSE**

Allows the application to take control of the system resources associated with the special file.

### Special Files That Can Be Inherited

The `tpf_fork` function permits a child process to inherit file descriptors from its parent process. When a file descriptor is inherited, its device driver functions can be called by multiple entry control blocks (ECBs) that are running concurrently on multiple I-streams using the same **filedata** object.

A file descriptor is inherited only if:

- Its file state data does not reference any ECB private storage, such as a noncommon core block or the ECB heap.
- All of its device driver functions can run concurrently on more than one I-stream.

For example, the `/dev/null` device driver used in the `TPF_FSDD_OPEN` function can be inherited because:

- It is stateless.
- It can run concurrently on multiple I-streams.

A file descriptor is inherited only if the UDDTBL device driver entry referenced by the file descriptor has the `TPF_SPIF_INHERITABLE` bit in its `tpf_spif_flags` member set to ON. If this flag is set to OFF, file descriptors that refer to the device driver will not be inherited by child processes.

### Application Control over Special File System Resources

There are conditions when it is desirable for an application to control special file resources rather than being managed by the device driver (for example, not closing a socket when there is no ECB accessing the socket). The `/dev/tpf.socket.file` special file allows you to treat a socket in the TPF system as if it is a regular file. However, one of the characteristics of a regular file is that it is closed automatically when all of the processes that have access to its open file descriptor have exited. In the TPF system, a socket can remain open even if no ECB has access to it; for example, this allows an ECB to pass a socket to a created ECB.

If the `TPF_SPIF_NODDCLOSE` flag in the `tpf_spif_flags` field for the UDDTBL device driver entry referenced by the file descriptor is set ON, the application can bypass the `TPF_FSDD_CLOSE` -type device driver function by calling the control open file descriptors (`fcntl`) function to set the `O_TPF_NODDCLOSE` open file description flag to ON. When the `TPF_SPIF_NODDCLOSE` flag in `tpf_spif_flags` field is set to OFF, attempting to set the `O_TPF_NODDCLOSE` open file description flag has no effect.

The following example shows how to open a socket as a special file without causing the socket to be closed when its file descriptor is closed.

```
int socfiledes;
pid_t childpid;
/*****
/* Open the socket and the socket special file.          */
*****/
char socfilename[80];
int socdes = socket( /* parameters specifying socket */);
sprintf(socfilename, "/dev/tpf.socket.file/%X", socdes);
socfiledes = open(socfilename, O_RDWR);

/*****
/* Take responsibility for the underlying socket from the file */
/* descriptor.                                                */
*****/
fcntl(socfiledes, F_SETFL,
      O_TPF_NODDCLOSE | fcntl(socfiledes, F_GETFL));

/*****
/* Pass the socket file descriptor to a child process by      */
/* inheritance through tpf_fork().                            */
*****/
childpid = tpf_fork( /* parameters specifying child process */ );

/*****
/* Close the file descriptor but not the underlying socket.   */
*****/
close(socfiledes);
```

---

## Device Drivers and Commit Scopes

All operations on the file system and on regular files are atomic. In effect, each file system API function runs in its own unique commit scope. The device driver functions, on the other hand, run in the same commit scope as the caller of the file system API. Therefore, it is up to the device driver to determine whether it runs as part of the application commit scope, if it suspends the application's commit scope, or if commit scopes are not relevant to it. You can write a device driver so that updates to the special files that it controls are part of a commit scope that encompasses a transaction or are like atomic updates to regular files.

---

## User-Defined Device Driver Functions

User-defined device driver functions are functions that users provide and the TPF system calls.

Types of user-defined device driver functions are defined in `c$spif.h`. To add a user-defined device driver, you must add a function of each of these types to the COMX library and add an entry initialized with pointers to those functions to the user device driver interface table (UDDTBL).

The following sections explain each type of function. The examples under these types use a device driver for the null file (`/dev/null`). The null file always succeeds

on writes (discarding the output data), is always at the end-of-file on reads, and always has a size of 0.

## TPF\_FSDD\_APPEND–Write Beginning at the End of a File

This type of function is specified as part of the file system device driver interface and is called by TPF file system C functions that require a physical append to a file device to append output data to an open special file.

### Format

```
typedef long TPF_FSDD_APPEND(void *buffer, long data_size, int noblock,
                             const TPF_FSDD_FILEDATA *filedata);
```

#### buffer

The address of the data buffer from which the output data will be appended. The range of addresses **buffer** through **buffer + data\_size – 1** is not verified to be addressable.

#### data\_size

The size of the output data to be appended to file. **data\_size** ranges from 1 to 2 147 483 647 ( $2^{31} - 1$ ) inclusive.

#### noblock

Nonblocking append request indicator.

For files that support nonblocking writes and cannot accept data immediately, you can write code to do the following:

- If **noblock** equals zero, the function blocks the process until the data can be appended.
- If **noblock** does not equal zero, the function does not block the process. If some data can be appended without blocking the process, the function appends what it can and returns the number of bytes appended. Otherwise, it sets `errno` to `EAGAIN` and returns the value `-1`.

#### filedata

The address of the file data object returned by the `TPF_FSDD_OPEN`-type device driver function for the special file being appended.

### Normal Return

The number of bytes appended from **buffer**, from 1 to **data\_size** inclusive.

**Note:** When zero bytes of data are requested to be appended, TPF file system C functions return zero without calling this device driver function.

### Error Return

`-1` means an error during the append operation. The device driver should also set `errno` to indicate the type of error.

### Programming Considerations

- The `TPF_FSDD_APPEND`-type device driver function is only called for open instances of files that were successfully opened by the corresponding `FSDD_OPEN`-type device driver function. More than one instance of a single file can be open at the same time.
- The `TPF_FSDD_APPEND`-type device driver function is never called for instances of files that are open with read-only access.
- If the capacity of the file is exceeded and no data is appended, the device driver should return `-1` and set `errno` to `EFBIG`; the device driver should never return `0`.

- A return value greater than 0 but less than **data\_size** indicates one of the following conditions:
  - The capacity of the file was too small to contain all the requested data.
  - Less than **data\_size** bytes of data could be appended to file without blocking for a nonblocking append request.
  - Only part of the requested data was successfully appended to file before an error prevented the remaining data from being appended. Here, the error is reported if it persists to the next append, preventing any data from being appended to the special file, at which time this device driver function returns **-1** and sets **errno**.
- If no data can be immediately appended to satisfy a nonblocking append request, the device driver should set **errno** to **EAGAIN** and return **-1**.
- If there is an error in the source of data or the device driver that prevents a write request from writing any valid data, the device driver should set **errno** to indicate the type of error and return **-1**.

## Examples

The following example is the append device driver interface function for the null file (**/dev/null**).

```
#include <c$spif.h> /* Device driver interface */

/*****
/* The null_append() function appends to a null file. It always
/* succeeds.
*****/
long null_append(void *data, long data_size, int non_block,
                 const TPF_FSDD_FILEDATA *filedata)
{
    return data_size;
}
```

## Related Information

- “assert—Verify Condition or Print Diagnostic Message” on page 15
- “close—Close a File” on page 44
- “fclose—Close File Stream” on page 127
- “fflush—Write Buffer to File” on page 144
- “fprintf, printf, sprintf—Format and Write Data” on page 201
- “fputc—Write a Character” on page 209
- “fputs—Write a String” on page 211
- “freopen—Redirect an Open File” on page 215
- “fseek—Change File Position” on page 226
- “fsetpos—Set File Position” on page 228
- “fsync—Write Changes to Direct Access Storage” on page 232
- “fwrite—Write Items” on page 239
- “TPF\_FSDD\_OPEN—Open a File” on page 1422
- “perror—Write Error Message to Standard Error Stream” on page 389
- “fprintf, printf, sprintf—Format and Write Data” on page 201
- “putc, putchar—Write a Character” on page 400
- “puts—Put String to Standard Output Stream” on page 402
- “vfprintf—Format and Print Data to a Stream” on page 678
- “vprintf—Format and Print Data to stdout” on page 680

## TPF\_FSDD\_APPEND

- “write—Write Data to a File Descriptor” on page 696.



## TPF\_FSDD\_CLOSE—Close a File

This type of function is specified as part of the file system device driver interface and is called by the `close`, `fclose`, and `freopen` functions to close a special file. Open files are also closed when a process ends.

### Format

```
typedef int TPF_FSDD_CLOSE(const TPF_FSDD_FILEDATA *filedata);
```

#### **filedata**

The address of the file data object returned by the `TPF_FSDD_OPEN`-type device driver function for the special file being closed.

### Normal Return

0        The special file has been closed as requested.

### Error Return

−1       The special file could not be closed as requested. The device driver should set `errno` to an appropriate error code.

## Programming Considerations

- The `TPF_FSDD_CLOSE`-type device driver function is only called once for each instance of a file that has been successfully opened by the corresponding `FSDD_OPEN`-type device driver function. More than one instance of a single file can be open at the same time.
- After an instance of an open file has been closed, whether successfully or not, no more device driver functions will be called against it.
- An `TPF_FSDD_CLOSE`-type device driver function must free any resources referenced by the **filedata** object.

## Examples

The following example is the close device driver interface function for the null file (`/dev/null`).

```
#include <c$spif.h> /* Device driver interface */

/*****
/* The null_close() function is a NOP.
*****/
void null_close(const TPF_FSDD_FILEDATA *filedata) {}
```

## Related Information

- “abort—Terminate Program Abnormally” on page 7
- “close—Close a File” on page 44
- “exit—Exit an ECB” on page 115
- “fclose—Close File Stream” on page 127
- “freopen—Redirect an Open File” on page 215
- “TPF\_FSDD\_OPEN—Open a File” on page 1422.

## TPF\_FSDD\_GET–Read From a File

This type of function is specified as part of the file system device driver interface and is called by TPF file system C functions that require a physical read from a file device to get input data from an open special file.

### Format

```
typedef long TPF_FSDD_GET(void *buffer, long buffer_size, long position,  
int noblock, const TPF_FSDD_FILEDATA *filedata);
```

#### **buffer**

The address of the data buffer into which the input data will be read. **buffer** is not a null pointer, but the range of addresses **buffer** through **buffer + buffer\_size – 1** is not verified to be either addressable or writable.

#### **buffer\_size**

The size of the input data buffer for the file. **buffer\_size** ranges from 1 to 2 147 483 647 ( $2^{31} - 1$ ) inclusive.

#### **position**

The logical position in the file from which the first byte of data will be read. **position** ranges from 0 to 2 147 483 647 ( $2^{31} - 1$ ) inclusive. The sum of **buffer\_size + position** is guaranteed to range from 1 to 2 147 483 648 ( $2^{31}$ ) inclusive.

#### **noblock**

Nonblocking read request indicator.

For files that support nonblocking reads and cannot accept data immediately, you can write code to do the following:

- If **noblock** equals zero, the function blocks the process until the data can be read.
- If **noblock** does not equal zero, the function does not block the process. If some data can be read without blocking the process, the function reads what it can and returns the number of bytes read. Otherwise, it sets `errno` to `EAGAIN` and returns the value `-1`.

#### **filedata**

The address of the file data object returned by the `TPF_FSDD_OPEN`-type device driver function for the special file being read.

### Normal Return

The number of bytes read into **buffer**, from 0 to **buffer\_size** inclusive.

### Error Return

`-1` means there was an error during the read operation. The device driver should also set `errno` to specify the type of error. The contents of **buffer** cannot be predicted.

### Programming Considerations

- The `TPF_FSDD_GET`-type device driver function is called only for open files that were successfully opened by the corresponding `TPF_FSDD_OPEN`-type device driver function. More than one instance of a single file can be open at the same time.
- The `TPF_FSDD_GET`-type device driver function is never called when a file is opened with write-only access.
- A return value of 0 means that the end of the file was reached.

**Note:** The Portable Operating System Interface for Computing Environments (POSIX) does not consider an end-of-file condition to be an error condition. The POSIX read function returns 0 bytes if an attempt is made to read at a position beyond the end of the file, and does not set any error flag. The standard C stream input/output (I/O) functions (in `stdio.h`) interpret zero bytes read as the EOF condition.

- A return value greater than 0 but less than **buffer\_size** specifies one of the following conditions:
  - There were less than **buffer\_size** bytes of data between **position** and the logical end of data.
  - There were less than **buffer\_size** bytes of data immediately available for a nonblocking read request.
  - Only part of the requested data was successfully read into **buffer** before an error prevented the remaining data from being read. Here, the error condition would be raised only if it persisted to the next read, preventing any bytes of data from being returned, at which time the device driver should return `-1` and set `errno`.
- If there is no data immediately available to satisfy a nonblocking read request, the device driver should set `errno` to `EAGAIN` and return `-1`.
- If there is an error in the source of data or the device driver that prevents a read request from returning any valid data, the device driver should set `errno` to specify the type of error and return `-1`.

## Examples

The following example is the get device driver interface function for the null file (`/dev/null`).

```
#include <c$spif.h> /* Device driver interface */

/*****
/* The null_get() always returns 0 (signifying, EOF).          */
*****/
long null_get(void *buffer, long buffer_size, long position,
              int non_block, const TPF_FSDD_FILEDATA *filedata)
{
    return 0;
}
```

## Related Information

- “fgetc—Read a Character” on page 146
- “fgets—Read a String from a Stream” on page 150
- “fread—Read Items” on page 213
- “fscanf, scanf, sscanf—Read and Format Data” on page 217
- “getc, getchar—Read a Character from Input Stream” on page 246
- “gets—Obtain Input String” on page 273
- “TPF\_FSDD\_OPEN—Open a File” on page 1422
- “read—Read from a File” on page 410
- “fscanf, scanf, sscanf—Read and Format Data” on page 217.

## TPF\_FSDD\_OPEN–Open a File

This type of function is specified as part of the file system device driver interface. This function is called by the `creat`, `fopen`, `freopen`, and `open` functions to make a special file accessible to the TPF file system.

### Format

```
typedef int TPF_FSDD_OPEN(int minordevice, char *location,
                          int opentype, TPF_FSDD_FILEDATA *filedata,
                          Ino_t inodeIno);
```

#### minordevice

The minor device number associated with a special file, defined by the `mknod` library function call that created the special file. **minordevice** ranges from 0 to 65 535 ( $2^{16} - 1$ ) inclusive. The meaning of the minor device number depends on the specific device driver.

#### location

The location information included in the tail of the pathname string passed to the `creat`, `fopen`, `freopen`, or `open` function for a special file. For example, if the special file "my.special.file" is opened as:

```
FILE *a_stream = fopen("my.special.file/at.XYZ", "r+b");
```

the pathname tail "at.XYZ" will be passed to this device driver function.

**location** points to a heap block that contains a '\0'-terminated character array. It is the responsibility of the device driver to free this block. The meaning of the location information depends on the specific device driver.

#### opentype

The indicator to open for read, write, or read and write access. The value of **opentype** is one of `O_WRONLY` (write only), `O_RDONLY` (read only), or `O_RDWR` (read and write access), defined in `fcntl.h`.

**Note:** `O_RDWR` is equal to `O_WRONLY | O_RDONLY`

#### filedata

The address of an object of type `TPF_FSDD_FILEDATA`, defined in `c$spif.h` as:

```
typedef struct TPF_FSDD_FILEDATA {
    long filedata_length;
    void * filedata_state_ptr;
} TPF_FSDD_FILEDATA;
```

This object can be used by the device driver to save state information between calls to the various device driver functions. It is passed to the `TPF_FSDD_OPEN`-type device driver function with `filedata_length` initialized to zero and `filedata_state_ptr` initialized to a NULL pointer. The `TPF_FSDD_OPEN`-type device driver function can store any file state information in this object. All other device driver functions can read, but cannot modify this object.

Each device driver function, which is called on a special file from the time it is opened to the time it is closed, will be passed the same `TPF_FSDD_FILEDATA` object that was passed to the `TPF_FSDD_OPEN`-type device driver function that opened the file. The actual types and values of the data referenced by this object are known only to the specific device driver and do not change after they have been set by the `TPF_FSDD_OPEN`-type device driver function.

**inodeIno**

The i-node ordinal number of type `Ino_t`, as defined in `sys/stat.h`, that has a one-to-one correspondence to the opened file.

**Normal Return**

- 0** The special file has been opened as requested.

**Error Return**

- 1** The special file could not be opened as requested. The device driver should set `errno` to an appropriate error code.

**Programming Considerations**

- The `TPF_FSDD_OPEN`-type device driver function can be called any number of times for the same file, regardless of whether previous instances of the same opened file have been closed. For example, the following program fragment gives access to "my.special.file" as three separate streams simultaneously: `FILE *fp1 = fopen("my.special.file", "r"); FILE *fp2 = fopen("my.special.file", "w"); FILE *fp3 = fopen("my.special.file", "a");`
- When a `TPF_FSDD_OPEN`-type device driver function returns 0 (the special file was opened successfully), a matching `TPF_FSDD_CLOSE`-type device driver function will be called unless the entry control block (ECB) exits because of a system error. The matching `TPF_FSDD_CLOSE`-type device driver function must free any resources referenced by the **filedata** object.
- When a `TPF_FSDD_OPEN`-type device driver function returns -1 (the special file was not opened successfully), do not leave resources referenced by the **filedata** object that need to be freed because no other device driver functions will be called for a file that fails to open.
- When a `TPF_FSDD_OPEN`-type device driver function returns -1, it should also set `errno` to specify the type of error.

**Examples**

The following example is the open device driver interface function for the null file (`/dev/null`).

```
#include <c$spif.h> /* Device driver interface */
#include <errno.h> /* errno, ENOTDIR */
#include <stdlib.h> /* free() */
#include <sys/types.h> /* Ino_t type */

/*****
/* The null_open() function opens a null file. It only fails if
/* location data is passed to it (it is not a directory); otherwise
/* it always succeeds.
*****/
int null_open(int minor_device, char *location, int open,
              TPF_FSDD_FILEDATA *filedata, (Ino_t)0)
{
    if (location) /* Invalid path name. */
    {
        free(location);
        errno = ENOTDIR; /* I am not a directory. */
        return -1;
    }
    return 0;
}
```

## TPF\_FSDD\_OPEN

### Related Information

- “fopen–Open a File” on page 199
- “freopen–Redirect an Open File” on page 215
- “TPF\_FSDD\_CLOSE–Close a File” on page 1419
- “mkfifo–Make a FIFO Special File” on page 328
- “mknod–Make a Character Special File” on page 331
- “open–Open a File” on page 380.

## TPF\_FSDD\_POLL–Select or Poll an Open File Descriptor

This type of function is specified as part of the file system device driver interface. This function is called by the `select` function to determine if the open file descriptor is ready for reading, writing, or if any exception conditions are pending. The device driver is determined by the major and minor device type from the open file descriptor at open time. A file descriptor can be of any device type that is supported by the TPF system.

### Format

```
typedef int TPF_FSDD_POLL(struct fd_entry *fd_ptr,
                        TPF_FSDD_FILEDATA *fita_ptr);
```

#### **fd\_ptr**

A pointer to a file descriptor contained in the `fd_entry` structure defined in the `i$publ.h` header file. The information contained in the structure is used to reference system program interface fields contained in the `c$spif.h` header file.

#### **fita\_ptr**

The address of an object of type `TPF_FSDD_FILEDATA`, defined in `c$spif.h` as:

```
typedef struct TPF_FSDD_FILEDATA {
    long filedata_length;
    void * filedata_state_ptr;
} TPF_FSDD_FILEDATA;
```

The information needed by the device drivers is carried in this object.

### Normal Return

One of the following values:

- 0**      The file descriptor is ***not*** ready for reading or writing, and there are no exception conditions pending.
- 1**      The file descriptor is ready for reading or writing, or there are exception conditions pending.

### Error Return

A value of `-1`. The request to check the state of the open file descriptor could not be processed because of a possible permanent internal error. The device driver should set `errno` to an appropriate error code. Calling programs should examine this return environment variable for additional information.

### Programming Considerations

The `TPF_FSDD_POLL`-type device driver function can be called any number of times for the same open file descriptor. The file descriptor entry must be open and accessible to be able to use this device driver function.

### Examples

The following example is the poll device driver interface function for the null file (`/dev/null`).

```
#include <c$spif.h> /* Device driver interface */

/*****
/* The null_poll() function is a NOP.
*****/
```

## TPF\_FSDD\_POLL

```
int null_poll(struct fd_entry *fd_ptr, TPF_FSDD_FILEDATA *filedata)
{
    return 0;
}
```

## Related Information

- “select—Monitor Read, Write, and Exception Status” on page 450
- “TPF\_FSDD\_OPEN—Open a File” on page 1422
- “TPF\_FSDD\_POLL\_CLEAN—Select or Poll Cleanup of an Open File Descriptor” on page 1427.



## TPF\_FSDD\_POLL\_CLEAN–Select or Poll Cleanup of an Open File Descriptor

This type of function is specified as part of the file system device driver interface. This function is called by the `select` function to direct the device driver to remove any pending `wait_queue` elements. `Wait_queue` elements are used during `select` function processing to determine if the open file descriptor is ready for reading, writing, or if any exception conditions are pending. The device driver is determined by the major and minor device type from the open file descriptor at open time. A file descriptor can be of any device type that is supported by the TPF system.

### Format

```
typedef int TPF_FSDD_POLL_CLEAN(struct fd_entry *fd_ptr,
                                TPF_FSDD_FILEDATA *fita_ptr);
```

#### **fd\_ptr**

A pointer to a file descriptor contained in the `fd_entry` structure defined in the `i$pwbl.h` header file. The information contained in the structure is used to reference system program interface fields contained in the `c$spif.h` header file.

#### **fita\_ptr**

The address of an object of type `TPF_FSDD_FILEDATA`, defined in `c$spif.h` as:

```
typedef struct TPF_FSDD_FILEDATA {
    long filedata_length;
    void * filedata_state_ptr;
} TPF_FSDD_FILEDATA;
```

The information needed by the device drivers is carried in this object.

### Normal Return

One of the following values:

- 0**      The file descriptor is **not** ready for reading or writing, and there are no exception conditions pending.
- 1**      The file descriptor is ready for reading or writing, or there are exception conditions pending.

### Error Return

A value of `-1`. The request to check the state of the open file descriptor could not be processed because of a possible permanent internal error. The device driver should set `errno` to an appropriate error code. Calling programs should examine this return environment variable for additional information.

### Programming Considerations

The `TPF_FSDD_POLL_CLEAN`-type device driver function can be called any number of times for the same open file descriptor. The file descriptor entry must be open and accessible to be able to use this device driver function.

### Examples

The following example is the poll cleanup device driver interface function for the null file (`/dev/null`).

```
#include <c$spif.h> /* Device driver interface */
```

```
/* ***** */
/* The null_poll_clean() function is a NOP.          */
```

## TPF\_FSDD\_POLL\_CLEAN

```
/*  
int null_poll_clean(struct fd_entry *fd_ptr, TPF_FSDD_FILEDATA *filedata)  
{  
    return 0;  
}
```

## Related Information

- “select—Monitor Read, Write, and Exception Status” on page 450
- “TPF\_FSDD\_POLL—Select or Poll an Open File Descriptor” on page 1425
- “TPF\_FSDD\_OPEN—Open a File” on page 1422.

## TPF\_FSDD\_PUT–Write to a File

This type of function is specified as part of the file system device driver interface and is called by TPF file system C functions that require a physical write to a file device to put output data to an open special file. When a process ends, any data remaining in file stream buffers is flushed to the file as part of the process of closing the open files.

### Format

```
typedef long TPF_FSDD_PUT(void *buffer, long data_size, long position,
    int noblock, const TPF_FSDD_FILEDATA *filedata);
```

#### buffer

The address of the data buffer from which the output data will be written. The range of addresses **buffer** through **buffer + data\_size – 1** is not verified to be addressable.

#### data\_size

The size of the output data to be written to file. **data\_size** ranges from 1 to 2 147 483 647 ( $2^{31} - 1$ ) inclusive.

#### position

The logical position in the file to which the first byte of data will be written. **position** ranges from 0 to 2 147 483 647 ( $2^{31} - 1$ ) inclusive. The sum of **data\_size + position** is guaranteed to range from 1 to 2 147 483 648 ( $2^{31}$ ) inclusive. If **position** is greater than the current logical file size and the put operation is successful, the bytes between the old end of data and the new data (logically) contain \0.

#### noblock

Nonblocking write request indicator.

For files that support nonblocking writes and cannot accept data immediately, you can write code to do the following:

- If **noblock** equals zero, the function blocks the process until the data can be written.
- If **noblock** does not equal zero, the function does not block the process. If some data can be written without blocking the process, the function writes what it can and returns the number of bytes written. Otherwise, it sets `errno` to EAGAIN and returns the value –1.

#### filedata

The address of the file data object returned by the TPF\_FSDD\_OPEN-type device driver function for the special file being written.

### Normal Return

The number of bytes written from **buffer**, from 1 to **data\_size** inclusive.

**Note:** When zero bytes of data are requested to be written, TPF file system C functions return zero without calling this device driver function.

### Error Return

–1 means there was an error during the write operation. The device driver should also set `errno` to specify the type of error.

## Programming Considerations

- The TPF\_FSDD\_PUT-type device driver function is called only for open instances of files that were successfully opened by the corresponding FSDD\_OPEN-type device driver function. More than one instance of a single file can be open at the same time.
- The TPF\_FSDD\_PUT-type device driver function is never called for instances of files that are open with read-only access.
- If the capacity of the file is exceeded and no data is written, the device driver should return `-1` and set `errno` to `EFBIG`; the device driver should never return `0`.
- A return value greater than `0` but less than **data\_size** specifies one of the following conditions:
  - The capacity of the file was too small to contain all the requested data.
  - Less than **data\_size** bytes of data that could be written without blocking for a nonblocking write request.
  - Only part of the requested data was successfully written to file before an error prevented the remaining data from being written. Here, the error condition is reported on the next write (if it persists and prevents any data from being written), at which time the device driver returns `-1` and sets `errno`.
- If no data can be immediately written to satisfy a nonblocking write request, the device driver should set `errno` to `EAGAIN` and return `-1`.
- If there is an error in the source of data or the device driver that prevents a write request from writing any valid data, the device driver should set `errno` to indicate the type of error and return `-1`.
- If the new data is written beyond the previous end of data in the file, any intervening bytes should be (logically) written as `\0`.

## Examples

The following example is the put device driver interface function for the null file (`/dev/null`).

```
#include <c$spif.h> /* Device driver interface */

/*****
/* The null_put() function writes to a null file. It always
/* succeeds.
*****/
long null_put(void *data, long data_size, long position, int non_block,
              const TPF_FSDD_FILEDATA *filedata)
{
    return data_size;
}
```

## Related Information

- “assert—Verify Condition or Print Diagnostic Message” on page 15
- “close—Close a File” on page 44
- “fclose—Close File Stream” on page 127
- “fflush—Write Buffer to File” on page 144
- “fprintf, printf, sprintf—Format and Write Data” on page 201
- “fputc—Write a Character” on page 209
- “fputs—Write a String” on page 211
- “freopen—Redirect an Open File” on page 215
- “fseek—Change File Position” on page 226

- “fsetpos–Set File Position” on page 228
- “fsync–Write Changes to Direct Access Storage” on page 232
- “fwrite–Write Items” on page 239
- “TPF\_FSDD\_OPEN–Open a File” on page 1422
- “perror–Write Error Message to Standard Error Stream” on page 389
- “fprintf, printf, sprintf–Format and Write Data” on page 201
- “putc, putchar–Write a Character” on page 400
- “puts–Put String to Standard Output Stream” on page 402
- “vfprintf–Format and Print Data to a Stream” on page 678
- “vprintf–Format and Print Data to stdout” on page 680
- “write–Write Data to a File Descriptor” on page 696.

## TPF\_FSDD\_RESIZE–Change the Size of a File

This type of function is specified as part of the file system device driver interface and is called by `fopen`, `ftruncate`, and `open` to change the size of the special file. If the file is extended beyond its current size, the extended part will (logically) contain `\0`.

### Format

```
typedef int TPF_FSDD_RESIZE(long size,
                           const TPF_FSDD_FILEDATA *filedata);
```

#### size

The new size of the special file. **size** ranges from 0 to 2 147 483 647 ( $2^{31} - 1$ ). If **size** is larger than the previous size of the file, the extended part will (logically) contain `\0`.

#### filedata

The address of the file data object returned by the `TPF_FSDD_OPEN`-type device driver function for the special file being resized.

### Normal Return

0        The size of the file was changed to **size**.

### Error Return

–1       The size of the file was not changed.

## Programming Considerations

- The `TPF_FSDD_RESIZE`-type device driver function is only called for open instances of files that were successfully opened by the corresponding `FSDD_OPEN`-type device driver function. More than one instance of a single file can be open at the same time.
- If the size of the file cannot be changed to the requested **size**, it should not be changed, the function should return `–1`, and `errno` should be set to indicate the type of error.
- If the size of a device cannot be determined or if it cannot be changed, use the following implementation of this function:

```
#include <c$spif.h>
#include <errno.h>

int device_type_resize(long new_size,
                      const TPF_FSDD_FILEDATA *state_ptr)
{
    errno = EINVAL;
    return -1;
}
```

## Examples

The following example is the set size device driver interface function for the null file (`/dev/null`).

```
#include <c$spif.h> /* Device driver interface */
#include <errno.h>  /* errno, EINVAL */

/*****
/* The null_resize() function always fails (the size of a null file
/* is always zero).
*****/
```

```
int null_resize(long new_size, const TPF_FSDD_FILEDATA *filedata)
{
    errno = EINVAL;
    return -1;
}
```

## Related Information

- “creat—Create a New File or Rewrite an Existing File” on page 54
- “fopen—Open a File” on page 199
- “ftruncate—Truncate a File” on page 237
- “TPF\_FSDD\_OPEN—Open a File” on page 1422
- “open—Open a File” on page 380.

## TPF\_FSDD\_SIZE–Get the Size of a File

This type of function is specified as part of the file system device driver interface and is called by `fcntl`, `fseek`, and `lseek` to determine the size of an open special file.

### Format

```
typedef long TPF_FSDD_SIZE(const TPF_FSDD_FILEDATA *filedata);
```

#### **filedata**

The address of the file data object returned by the `TPF_FSDD_OPEN`-type device driver function for the special file being sized.

### Normal Return

The size of the special file, in bytes, or `-1` if the size cannot be determined.

### Error Return

Not applicable.

## Programming Considerations

- The `TPF_FSDD_SIZE`-type device driver function is called only for open instances of files that were successfully opened by the corresponding `FSDD_OPEN`-type device driver function. More than one instance of a single file can be open at the same time.
- For devices that do not have an inherent file size (for example, terminals or printers) or for which the size of data cannot be determined (for example, communications links), this function should return `-1`.
- This function is used to implement the seek functions that change the file offset. Device drivers for special files that have meaningful offsets must implement this function in a way that is consistent with the device. If the size or file offset is not meaningful, use the following implementation of this function:

```
#include <c$spif.h>
#include <errno.h>

long device_type_size(const TPF_FSDD_FILEDATA *state_ptr)
{
    errno = EINVAL;
    return -1;
}
```

## Examples

The following example is the size device driver interface function for the null file (`/dev/null`).

```
#include <c$spif.h> /* Device driver interface */
/*****
/* The null_size() function always returns a size of 0.
*****/
long null_size (const TPF_FSDD_FILEDATA *filedata)
{
    return 0;
}
```

## Related Information

- “`fcntl`—Control Open File Descriptors” on page 129
- “`fseek`—Change File Position” on page 226



- “TPF\_FSDD\_OPEN–Open a File” on page 1422
- “lseek–Change the Offset of a File” on page 315.

## TPF\_FSDD\_SYNC–Synchronize the File Data

This type of function is specified as part of the file system device driver interface and is called by the `fsync` function to synchronize any data that may be buffered by the device driver or by the TPF system.

### Format

```
typedef long TPF_FSDD_SYNC(const TPF_FSDD_FILEDATA *filedata);
```

#### filedata

The address of the file data object returned by the `TPF_FSDD_OPEN`-type device driver function for the special file being synchronized.

### Normal Return

**0** The data was synchronized successfully.

### Error Return

**–1** An error occurred during the synchronization operation. The device driver should also set `errno` to indicate the type of error.

## Programming Considerations

- This device driver function is optional. If it is not present (meaning that its slot in the device driver table in segment `CDDTBL` or `UDDTBL` is 0), the `fsync` function returns 0 (success). Code the file system function only if there is system-level buffering that is under application control.
- The `TPF_FSDD_SYNC`-type device driver function is called only for instances of files that were successfully opened by the corresponding `FSDD_OPEN`-type device driver function. More than one instance of a single file can be open at the same time.

## Examples

The following example is the synchronization device driver interface function for the null file (`/dev/null`).

```
#include <c$spif.h> /* Device driver interface */

/*****
/* The null_sync() function synchronizes a null file. It always
/* succeeds. Note that this function does not need to be specified
/* for the null file device driver because the default behavior when
/* no TPF_FSDD_SYNC-type function is specified also is to always
/* succeed.
*****/
long null_sync(const TPF_FSDD_FILEDATA *filedata)
{
    return 0;
}
```

## Related Information

- “`fsync`–Write Changes to Direct Access Storage” on page 232
- “`TPF_FSDD_OPEN`–Open a File” on page 1422.

---

## Appendix F. GNTAGH User's Guide

The GNTAGH program is a global support utility that is designed to assist C language support users in creating and maintaining TPF globals. The GNTAGH program automates what would otherwise be a tedious manual task of creating C identifiers that map to assembler labels for TPF globals.

The GNTAGH program processes assembler output to produce a C header file (c\$globz.h) that defines tags for the TPF C language interface to globals; it takes an HLASM SYSADATA file as its input.

This appendix describes how to use the GNTAGH program on both the VM/CMS and MVS operating systems.

---

### C Global Tagnames

The purpose of the GNTAGH program is to create C language global tagnames that correspond to assembler global tags. Global tagnames consist of unique 32-bit numbers that describe global attributes. (See "Format of Global Tags" on page 1441 for more information about global tags.) To create the C tagnames and make them available to your programs:

1. Assemble an assembler source program, producing an output file that describes all of the assembler tags to be converted to C tags.
2. Compile the GNTAGH program.
3. Run the GNTAGH program to produce the c\$globz.h header file, which contains the global tagnames.
4. Verify the keypointability attribute of all tagnames in the c\$globz.h header file that are listed as keypointable.
5. Include the c\$globz.h header file in all of your C functions and programs that require access to TPF globals.
6. Recompile all C programs that refer to TPF globals.

These tasks are discussed in greater detail in the following sections. Creating the c\$globz.h header file in tasks 1–3 is automated by the GENGLOBH exec on the VM/CMS system and by GLOBJCL on the MVS system.

---

### Program Parts List

The ACP.CSRCE.OL.RELv partitioned data set contains the GNTAGH program as well as some samples of source code that you need to run it. You must adapt this sample code to your environment. Data set members to be used with HLASM output are shown in Table 52.

*Table 52. ACP.CSRCE.OL.RELv Data Set Members for Converting HLASM Output*

Member Name	Description
GLOBALS	This assembler source file contains two calls to the GLOBZ macro. The first GLOBZ invocation defines global area 1 (REGR= parameter). The second GLOBZ invocation defines global area 3 (REGS= parameter). This order must be maintained. <b>Note:</b> The GLOBALS program does not require user modification; however, the SYSADATA output from the its assembly is used as input to GNTAGH, so it should be reassembled (and GNTAGH rerun) whenever the contents of any of the global block DSECTs change.

Table 52. ACP.CSRCE.OL.RELvv Data Set Members for Converting HLASM Output (continued)

Member Name	Description
GLDEFH	This is a header file that contains constant definitions required by GNTAGH at compile time.
GNTAGH	This C source program contains the actual statements that create the c\$globz.h header file from HLASM SYSADATA input. It may require user modification, and it should be recompiled whenever the GLDEFH header file is modified.
GENGLOBH	This program is for VM/CMS installations only, and requires the IBM C/370 Compiler (5688-040) and IBM C/370 Library (5688-039), and their associated prerequisites. GENGLOBH contains the instructions necessary to reassemble GLOBALS, recompile GNTAGH, and generate the resulting c\$globz.h header file. GENGLOBH <b>requires</b> user modification, and is provided only as an example.
GLOBHJCL	This program is for MVS installations only, and requires the IBM C/370 Compiler (5688-040) and IBM C/370 Library (5688-039), and their associated prerequisites. GLOBHJCL contains the instructions necessary to reassemble GLOBALS, recompile GNTAGH, and generate the resulting c\$globz.h header file. GLOBHJCL <b>requires</b> user modification, and is provided only as an example.

The following two sections tell you how to create C global tagnames. An overview of this entire process is shown in Figure 3 on page 1441.

If you are using the MVS operating system, skip to “Creating or Updating C Globals on MVS” on page 1439.

---

## Creating or Updating C Globals on VM

You must complete the following tasks to create or update C language TPF globals on VM (steps 1–3 are automated by the GENGLOBH exec):

1. Assemble GLOBALS.

Specify the ADATA option (the cross reference is not required; cross reference information is ignored by the GNTAGH program).

The SYSADATA file that is produced will be used as input to the GNTAGH program. It must be on your A-disk; it must be accessible in disk search order for CMS.

2. Compile and link the GNTAGH program.

If your installation runs a non-standard global system in which the attributes of global blocks or areas are different from those defined by IBM, you must:

- a. Modify GLDEFH or GNTAGH as required

You must pay particular attention to the array of struct blk\_desc called block. This array is defined in the GNTAGH program. Each item in this array describes a global block and contains the following items, which **must** be initialized to constant values prior to compiling the GNTAGH program:

- Keypointability
- Subsystem commonality
- Global area of residence
- Global block name
- Number of keypointable records if a directory block.

The remainder of structure members are initialized at program run time and will be overwritten if initialized.

- b. Recompile the GNTAGH program.
3. Issue FILEDEF commands, then enter **GNTAGH** on the CMS command line.  
The file name of the assembler SYSADATA file must be associated with the DD name INFILE, and should be GLOBALS SYSADATA.  
Use C\$GLOBZ H for the DD name OUTFILE. DCB information does not have to be specified for VM/CMS because file type H defaults to fixed records of length 80.
4. Verify that all tagnames that are listed as keypointable in the c\$globz.h header file are actually keypointable. (The GNTAGH program cannot determine if certain TPF globals are keypointable, so the program assumes that they are.) Change all tagnames that are not correct.
5. Include the c\$globz.h header file in all of your C functions and programs that require access to TPF globals. Sample C\$GLOBZ H output is shown in “Sample Output” on page 1445.
6. Recompile all C programs that reference, either explicitly or implicitly, the c\$globz.h header file. (This is not necessary if you have been careful to ensure that global field displacements, lengths and characteristics have not been impacted or changed by the updates.)  
If any of a global tagname’s attributes change, you need to reconstruct the c\$globz.h header file so that C functions are accessing the new information. You must also recreate the c\$globz.h header file whenever any of the following events occur:
  - A new global field is inserted before existing fields
  - The length of an existing global field is changed
  - A keypointable global directory item is added, deleted or changed
  - An existing global field is deleted, changing displacement of remaining fields
  - A global field is moved from one block to another in order to change an attribute
  - A global field or record is added to the SIGT table, making it synchronizable
  - The role or attributes of a global block are changed
  - A new global block or area is added.

The following section is for MVS users only. To bypass this section, go to “Format of Global Tags” on page 1441.

---

## Creating or Updating C Globals on MVS

You must complete the following tasks to create or update C language TPF globals on MVS (steps 1–3 are automated by GLOBHJCL):

1. Assemble GLOBALS.  
Specify the ADATA option (the cross reference is not required; cross reference information is ignored by the GNTAGH program).  
The SYSADATA file that is produced will be used as input to the GNTAGH program. It may be kept in either a user-catalogued data set (under MVS/TSO), or in a temporary data set used with JCL (under MVS/JES).
2. Compile and link the GNTAGH program.  
If your installation runs a nonstandard global system in which the attributes of global blocks or areas are different from those defined by IBM, you must:

- a. Modify GLDEFH or GNTAGH as required.

You must pay particular attention to the array of struct `blk_desc` called `block`. This array is defined in the GNTAGH program. Each item in this array describes a global block, and contains the following items, which **must** be initialized to constant values before compiling the GNTAGH program:

- Keypointability
- Subsystem commonality
- Global area of residence
- Global block name
- Number of keypointable records if a directory block.

The remainder of structure members are initialized at program run time and will be overwritten if initialized.

- b. Recompile the GNTAGH program.

3. Run the GNTAGH program directly under TSO or by submitting a job to the job entry system (JES).

The HLASM SYSADATA output data set must be associated with the DD name INFILE.

Use ACP.CHDR.RELvv(C\$GLOBZ) for the DD name OUTFILE. Because the GNTAGH program has no DCB information coded for this file, you must provide it. The DCB information is required on either the JES3 DD JCL statement or in the TSO ALLOC statement. The following DCB specification is recommended:

```
DCB=(RECFM=F,LRECL=80)
```

4. Verify that all tagnames that are listed as keypointable in the `c$globz.h` header file are actually keypointable. (The GNTAGH program cannot determine if certain TPF globals are keypointable, so the program assumes that they are.) Change all tagnames that are not correct.
5. Include the `c$globz.h` header file in all of your C functions and programs that require access to TPF globals. Sample C\$GLOBZ output is shown in "Sample Output" on page 1445.
6. Recompile all C programs that refer to, either explicitly or implicitly, the `c$globz.h` header file. (This is not necessary if you have been careful to ensure that global field displacements, lengths and characteristics have not been impacted or changed by the updates.)

If any of a global tagname's attributes change, you need to reconstruct the `c$globz.h` header file so that C functions are accessing the new information.

You must also recreate the `c$globz.h` header file whenever any of the following events occur:

- A new global field is inserted before existing fields
- The length of an existing global field is changed
- A keypointable global directory item is added, deleted or changed
- An existing global field is deleted, changing displacement of remaining fields
- A global field is moved from one block to another, in order to change an attribute
- A global field or record is added to the SIGT table, making it synchronizable
- The role or attributes of a global block are changed
- A new global block or area is added.

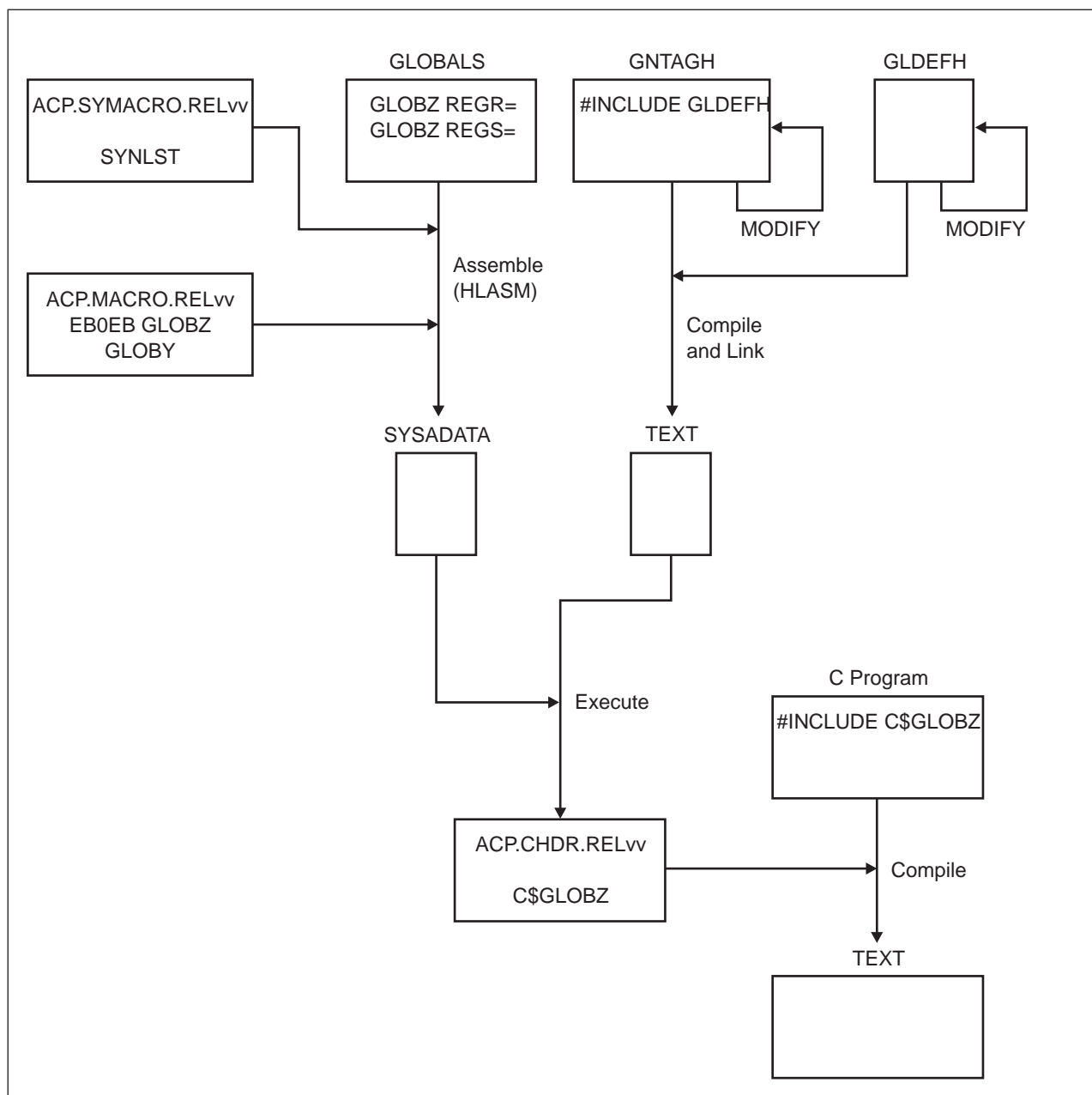


Figure 3. Creating C Language Global Tagnames

## Format of Global Tags

Every global tagname is assigned a unique 32-bit number that describes its displacement in a global area, its length as defined, the number of the global area in which it resides, and characteristics of online handling such as keypointability, subsystem commonality, or uniqueness. The format of the global tagname is shown in Table 53.

Table 53. Format of Global Tag Definitions

Bits	Description
0-3	Reserved, and set to B'0000'.

Table 53. Format of Global Tag Definitions (continued)

Bits	Description
4-15	Displacement of the item in its global area (a value between X'000' and X'FFF').
16-23	<ul style="list-style-type: none"> <li>Field length, if tagname refers to a field</li> <li>Length of the directory, if tagname refers to a record</li> <li>The SIGT slot number of the field or record, if the field or record is synchronizable.</li> </ul>
24	<b>0</b> = not keypointable <b>1</b> = keypointable
25	<b>0</b> = not synchronizable <b>1</b> = synchronizable <b>Note:</b> This attribute is determined from the contents of the SYNLIST copy member. (This is produced in SIP Stage II, when SIP macro SKSYN is assembled.)
26	<b>0</b> = field <b>1</b> = record
27	Reserved for IBM use.
28	<b>0</b> = field or record is not SSU common <b>1</b> = field or record is SSU common
29	Reserved for customer use.
30-31	<b>1</b> = global area 1 <b>3</b> = global area 3

## Non-relocatable Symbols

The GNTAGH program does not convert non-relocatable symbols found in the GLOBALS assembly to global tags. Instead, these are defined in the `c$globz.h` header file with the same value to which they are equated to in GLOBALS. Non-relocatable symbols are usually defined like

```
@EXAMPLE EQU    X'80'
```

If your installation defines global record or field names as non-relocatable symbols, and you require GNTAGH to convert them to global tags, you can do so by compiling GNTAGH with the CONVNONRELOC macro defined, either by adding

```
DEFINE(CONVNONRELOC)
```

to the compiler parameter list, or by adding

```
#define CONVNONRELOC
```

to the `gldefh.h` header file.

## GNTAGH Messages

All non-header file output from the GNTAGH program is sent to SYSPRINT (console in CMS or TSO) and is in the following format:

```
GHTnnnnn ccccc Message text
```

where:



**nnnnn =**

A sequence number. Each message number has a unique sequence number associated with it, identifying the phase in which the error or information was generated.

**cccccc =**

Message class. There are three types of messages sent:

**INFO** Information only

**WARNING** Possible error condition

**FATAL** An error condition from which recovery is not possible. If a FATAL message has been received, the program will exit with a return code equal to the message sequence number, closing all open files where possible.

**Message text**

Each message class field is followed by text that describes the condition.

---

**00016**

**Severity:** FATAL

**Explanation:** Unable to open assembly SYSADATA file. This message occurs when the GLOBALS SYSADATA file cannot be opened in read-only mode.

---

**00017**

**Severity:** FATAL

**Explanation:** Unable to allocate working storage. This message occurs when the GNTAGH program is unable to allocate working storage for its internal processing.

---

**00018**

**Severity:** FATAL

**Explanation:** Unable to create temporary working file. This message occurs when the GNTAGH program is unable to create a scratch file for its internal processing.

---

**00020**

**Severity:** FATAL

**Explanation:** Unable to open output file. This message occurs when the c\$globz.h header file cannot be opened in write-only mode.

---

**00021**

**Severity:** FATAL

**Explanation:** Unrecognized COPY statement found before SYNLST. This message is received when another COPY statement precedes the COPY SYNLST statement in the SYSADATA file. This usually indicates that either the GLOBALS program has been altered or that the wrong SYSADATA file is being read.

---

**00022**

**Severity:** FATAL

**Explanation:** COPY SYNLST statement not found. This message is received when the GNTAGH program cannot locate the COPY SYNLST statement in the SYSADATA file. This usually indicates that either the GLOBALS program has been altered or that the wrong SYSADATA file is being read.

---

**00024**

**Severity:** FATAL

**Explanation:** Cannot find FLDCNT and/or RDCDCNT in SYNLST. This message is received when the SYNLST macro expansion does not contain the assembler GBLA variables FLDCNT and/or RDCDCNT, which define the number of synchronizable fields and record entries in the SIGT table. Check to see if SYNLST has been modified, and these variables removed.

---

**00030**

**Severity:** FATAL

**Explanation:** GLOBZ REGR or GLOBZ REGS statement not located or not in sequence. This message occurs when the SYSADATA file either does not contain the GLOBZ REGR= or GLOBZ REGS= statement, GLOBZ REGR= is not the first GLOBZ macro invoked, or GLOBZ REGS= is not the second GLOBZ macro invoked.

---

**00040**

**Severity:** FATAL

**Explanation:** Missing global block label. This message is received when one or more global block names, as defined in the "block" array in GNTAGH, have not been

located among the SYSADATA SYMBOL records.  
Check to make sure that:

1. There are no extraneous global block names defined in GNTAGH.
2. The proper MACLIBs were used for assembly.

---

**00050**

**Severity:** FATAL

**Explanation:** Error writing to output file. This message is received when an attempt to write an output line to C\$GLOBZ fails.

---

**00098**

**Severity:** WARNING

**Explanation:** Rejecting — unable to locate block. This message is received when a global tagname has been encountered whose displacement does not fall into any of the specified global blocks/areas. Check to make sure that GNTAGH code has not been altered in such a way that it cannot handle certain references.

---

**00099**

**Severity:** WARNING

**Explanation:** Rejecting — low definition line. This message is received when a global tagname has been encountered whose point of definition was not within the bounds established by the GLOBZ statements. The line number and tagname as received are displayed in the text of the message, and should be checked against the output shown in message 00200. When this message is received, check to make sure that the GNTAGH program has not been altered in such a way that it cannot reconcile certain line numbers.

---

**00200**

**Severity:** INFO

**Explanation:** This message identifies the assembly source lines where each of the required GLOBZ statements is located. It is needed only for diagnostic purposes and does not describe an error condition.

---

**00300**

**Severity:** INFO

**Explanation:** C\$GLOBZ H complete. This message is received at end-of-job when no errors have been encountered. The text of this message contains a count of the number of tags successfully converted into C #define format.

---

**00301**

**Severity:** WARNING

**Explanation:** Errors encountered. This message is received when at least one symbol name in global tagname format did not qualify for inclusion in the output header file. This message is sent at end-of-job instead of message 300 when errors have been encountered. Check your GLOBALS listing against the C\$GLOBZ file produced by this run to identify the error.

---

**00302**

**Severity:** INFO

**Explanation:** This message is a reminder that you must verify the keypointability attribute of all tagnames in the c\$globz.h header file that are listed as keypointable.

---

## GNTAGH Program Logic Flow

Open INFILE DD (GLOBALS LISTING) for read-only operations.

Open OUTFILE DD (C\$GLOBZ H) for write-only operations, destroying any existing copy of the file.

Open a scratch file for global symbols and their attributes.

Using SYSADATA SOURCE records:

1. Locate COPY SYNLST statement in macro expansion, then obtain names and slot numbers for all synchronizable fields and records.
2. Locate GLOBZ REGR= statement in open code, note line number.
3. Locate GLOBZ REGS= statement in open code, note line number.

Using SYSADATA SYMBOL records:

1. Locate all of the global block names, defined in the "block" array.
2. For each symbol beginning with an at sign (@),
  - a. Convert all occurrences of the "at" sign (@), "pound" sign (#) or "dollar" sign (\$) in the tagname to underscores (\_), and all uppercase letters to lowercase.
  - b. Call function "skip\_this\_tag", passing the converted tagname, to determine if the tag should be excluded.
  - c. For each non-excluded tagname, write a record to the scratch file including the tagname, its displacement, the line number in which it is defined, its length attribute, and a flag indicating whether or not it is relocatable.

Rewind the scratch file:

For each global tag record in the scratch file:

- If the symbol named in the record is relocatable:
  - From the line number of definition, determine which global area the tag lies in.
  - From the given displacement, determine which global block the tag lies in.
  - Use the characteristics of keypointability, subsystem commonality, displacement and length to calculate the unique number to be associated with the tagname.
  - If the tagname is in the array of synchronizable items, OR in the synchronizable bit and substitute the slot number for the item's length.
  - If global block of residence is GLOBA or GLOBY, test the displacement of the tagname—if within the range of the keypointable global directory items, OR on the "global record" indicator.
- Else,
  - The value of the #define will be just the displacement (i.e. the EQUated value).
- Write line to output file for #define term, using the generated number.

Close files and summarize execution statistics.

---

## Sample Output

Following is a sample of a header file that was produced by using the GNTAGH program:

```

/* C$GLOBZ.H - TPF Global Tag Name Constants
**
** Generated by GNTAGH: Mon Dec 13 14:40:26 1993
*/
#ifndef __global_tags__
#define __global_tags__
#define _alpha      0x04400201
#define _artcr      0x00c008a3
#define _brcpa      0x0248010b
#define _brcpb      0x0249010b
.
.
.
#define _x2tqcf      0x009c04a1
#define _x2trt      0x008804a1
#define _x2trtf      0x008c04a1
#endif
/* End of C$GLOBZ.H */

```

## References

*TPF System Installation Support Reference.*

---

## Appendix G. IPRSE - A Parser Utility for TPF Systems

IPRSE is a C utility that is used for matching parameter lists with grammars and returning the values of the parameters if they match. You code a function that defines a grammar to IPRSE for string parsing. The strings can be input commands or other similar application strings that need parsing. The parameters in the input string must follow the grammar rules for the parser.

To use the parser:

1. Define a grammar in a function; see “Defining a Grammar” on page 1448 that follows.
2. Call the following functions to activate the parser:
  - a. “IPRSE\_parse—Parse a Text String against a Grammar” on page 1456  
Matches the parameter list with the grammar and returns the values of the parameters and a return code as output.
  - b. A function that processes the parsed output.

For a complete, coded example showing the grammar, the parsing, and the call to a function to process the parsed output, see “Examples” on page 1458.

3. Understand the errors that can result from an incorrect grammar or an input string.

When an input string does not match the grammar, the parser issues online messages, which are controlled by an option specified on the IPRSE\_parse function. When an invalid grammar is detected, the system issues an OPR dump. For more information, see “Error Return” on page 1457.

---

### Parser Options That Affect How the Input String Matches the Grammar

There are four options that affect the way that the parser matches a grammar to an input string:

IPRSE\_NOSTRIC  
IPRSE\_STRICT  
IPRSE\_NOMIXED\_CASE  
IPRSE\_MIXED\_CASE

The IPRSE\_NOSTRIC option is compatible with the assembler parser (see the BPKDC and BPPSC macros in *TPF General Macros*) in terms of the characters that are used to separate parameters from each other, and to separate keywords from values. The IPRSE\_STRICT option restricts the number of characters that are interpreted as special characters, allowing more characters to be used for parameters (see “Special Characters for Input String Syntax” on page 1448). For example, using the IPRSE\_NOSTRIC option, the input string “a=b/c,d” is parsed into three separate tokens:

1. a=b
2. c
3. d

Using the IPRSE\_STRICT option, the same input is parsed as a single token.

The IPRSE\_NOMIXED\_CASE option is used when the input string contains only uppercase letters. If the input string can contain lower case letters you must specify the IPRSE\_MIXED\_CASE option.

You can specify these options when you call the `IPRSE_parse` function (see “`IPRSE_parse`—Parse a Text String against a Grammar” on page 1456). You also can specify these options at the beginning of the grammar (see “Specifying Parser Options in the Grammar” on page 1449). Options specified in the grammar override options specified when you call the `IPRSE_parse` function. The effects of these options are described in detail in the following sections.

---

## Special Characters for Input String Syntax

The following characters have special syntax definitions for input strings:

<b>blank ( )</b>	Token delimiter
<b>comma (,)</b>	Token delimiter ( <code>IPRSE_NOSTRICT</code> only)
<b>slash (/)</b>	Token delimiter ( <code>IPRSE_NOSTRICT</code> only)
<b>period (.)</b>	Special delimiter that ties 2 or more parameters together
<b>hyphen (-)</b>	Keyword delimiter
<b>equal sign (=)</b>	Keyword delimiter ( <code>IPRSE_NOSTRICT</code> only).

---

## Defining a Grammar

This section lists the syntax rules for defining a grammar to `IPRSE`.

The topics in this section are:

- “Special Characters for Grammar Syntax”
- “Specifying Parser Options in the Grammar” on page 1449
- “Parameters” on page 1449
  - “Positional Parameters” on page 1452
  - “Keyword Parameters” on page 1452
    - “Regular Keywords” on page 1453
    - “Self-defining Keywords” on page 1454
- “Translating Input String Values to Upper Case” on page 1454
- “Programming Considerations for Grammars” on page 1455
- “Examples of Grammar Definitions” on page 1455.

## Special Characters for Grammar Syntax

The following characters have special syntax definitions for grammars:

<b>blank ( )</b>	Parameter delimiter
<b>braces ({})</b>	Defines alternative parameters
<b>brackets ([])</b>	Defines optional parameters
<b>comma (,)</b>	Parameter delimiter
<b>slash (/)</b>	Parameter delimiter
<b>vertical bar ( )</b>	Delimiter for alternative parameters within { }
<b>parentheses (())</b>	Defines a list of parameters
<b>period (.)</b>	Special delimiter that ties 2 or more parameters together
<b>plus sign (+)</b>	Optional parameter character
<b>asterisk (*)</b>	Optional string or wildcard list
<b>hyphen (-)</b>	Keyword delimiter

<b>equal sign (=)</b>	Keyword delimiter
<b>less-than sign (&lt;)</b>	Grammar option delimiter, translate input value to upper case grammar token suffix (see "Translating Input String Values to Upper Case" on page 1454)
<b>greater-than sign (&gt;)</b>	Grammar option delimiter.

## Specifying Parser Options in the Grammar

At the beginning of the grammar you can specify the following parser options:

```
IPRSE_NOSTRIC
IPRSE_STRICT
IPRSE_NOMIXED_CASE
IPRSE_MIXED_CASE
```

Each option must be enclosed in angle brackets (<>), with no spaces between the angle brackets and the option, and all options must precede any other grammar tokens. For example, the following grammar specifies the IPRSE\_NOSTRIC and IPRSE\_MIXED\_CASE options:

```
const char grammar[] =
    "<IPRSE_NOSTRIC><IPRSE_MIXED_CASE>"
    "ZXXX NUMbers-d* LETters-c*";
```

The <tpfparse.h> header file defines the following symbolic names for grammar parser options:

```
#define IPRSE_STRICT_GRAMMAR      "<IPRSE_STRICT>"
#define IPRSE_NOSTRIC_GRAMMAR    "<IPRSE_NOSTRIC>"
#define IPRSE_MIXED_CASE_GRAMMAR "<IPRSE_MIXED_CASE>"
#define IPRSE_NOMIXED_CASE_GRAMMAR "<IPRSE_NOMIXED_CASE>"
```

Thus, the preceding grammar example can also be coded as follows:

```
const char grammar[] =
    IPRSE_NOSTRIC_GRAMMAR IPRSE_MIXED_CASE_GRAMMAR
    "ZXXX NUMbers-d* LETters-c*";
```

Parser options specified at the beginning of a grammar override any conflicting options specified in the IPRSE\_parse function options parameter. For example, in the following code fragment the IPRSE\_MIXED\_CASE option specified in the grammar (the second parameter) overrides the IPRSE\_NOMIXED\_CASE option specified in the options parameter (the fourth parameter):

```
int count = IPRSE_parse("a b c",          /* input string */
    "<IPRSE_MIXED_CASE> A B C",          /* grammar */
    &result,
    IPRSE_ALLOC | IPRSE_NOMIXED_CASE, /* options */
    error_header);
```

All grammar options must be specified at the beginning of the grammar; there are no default grammar options. If a grammar specifies two conflicting options (<IPRSE\_NOSTRIC> and <IPRSE\_STRICT>, or <IPRSE\_NOMIXED\_CASE> and <IPRSE\_MIXED\_CASE>), the last option specified overrides any previous ones.

## Parameters

In the grammar, there can be two types of parameters: positional and keyword. Any positional parameters must come before the keywords.

Input strings must either match characters or match character types, depending on the type of parameter in the grammar. In the grammar, characters are letters (A-Z)

or digits (0-9), and character types are a, c, d, u, w, x, \*, and +, which are described later in this section. The sections on positional parameters and keyword parameters explain how matching characters and character types are used by the grammar for the parameter.

The following are general rules for parameters in the grammar:

- Define optional parameters in left and right brackets: ([ ]).

Example:

Grammar: [C,A]

Input strings of C,A or nothing

- Define alternative parameters in left and right braces ({}), with a vertical bar (|) being used to delimit the choices. Exactly 1 parameter must be selected for the input string.

Example:

Grammar: {A|B}

Input strings must be either: A or B, but not both

- Use either a blank ( ), a comma (,), or a slash (/) to delimit parameters.

A period (.) can be used to delimit parameters that use character types.

The parser deletes multiple delimiters between parameters. The syntax does not include a null positional parameter.

Examples:

A B,C            contains 3 parameters A, B and C

A , B            contains 2 parameters A and B

A ,, B           contains 2 parameters A and B.

- A parameter:

- Can be abbreviated.

In the grammar, leading uppercase letters indicate that they must be matched in the input string, and trailing lowercase letters indicate that they are optional for the input string. If you are creating a grammar for a parameter with matching characters, always begin it with an uppercase letter. (In the input string, all letters must be entered in uppercase unless the IPRSE\_MIXED\_CASE option is specified.)

- Can be an alphanumeric character string of any length.

- Can be restricted to the following character types:

- a** Accepts one alphanumeric character, including uppercase letters (A-Z), digits (0-9), and underscores (\_). If the IPRSE\_MIXED\_CASE option is specified, **a** also accepts lowercase letters (a-z).
- c** Accepts one uppercase letter (A-Z). If the IPRSE\_MIXED\_CASE option is specified, **c** also accepts lower case letters (a-z).
- d** Accepts one decimal digit (0-9).
- n** Accepts one uppercase letter (A-Z), one decimal digit (0-9), or one of the “national” characters: at sign (@), pound sign (#) or dollar sign (\$). If the IPRSE\_MIXED\_CASE option is specified, **n** also accepts lowercase letters (a-z).
- u** Accepts one character of unrestricted type, that is, any character that does not have special meaning for input strings (see “Special Characters for Input String Syntax” on page 1448). The characters that have special meaning for input strings when the IPRSE\_NOSTRICT option is specified are: period (.), dash (-), equal sign (=), comma (,), slash (/), and spaces.



The characters that have special meaning for input strings when the IPRSE\_STRICT option is specified are: period (.), dash (-), and spaces.

Unlike **w** (see below), **u** treats asterisks (\*) the same as any other character; asterisks in the input string must match **us** in the grammar one-for-one.

- w** Accepts one alphanumeric character, the same as the **a** character type (with the IPRSE\_MIXED\_CASE option controlling whether or not **w** accepts lowercase letters). In addition, **w** can accept one asterisk (\*).

An asterisk (\*) in the input string behaves as a wildcard character. In the input string, one asterisk can match any number of **w** character types in the grammar. If **w** is coded in a grammar, it cannot be followed by a different character type.

- x** Accepts one hexadecimal digit (0-F or 0-f).

- \*** After a character type indicates that the character type can be optionally repeated as many times as needed. A **\*** cannot be followed by any other character type. If a grammar parameter begins with **\***, it matches character type **a**.

- +** After a character type indicates that the character type can be optionally repeated as many times as the plus sign (+) appears. A **+** cannot be followed by any other character type except for another **+**. If a grammar parameter begins with **+**, it matches character type **a**.

- Parameters can have lists as their values: a regular or wildcard list.

- A regular list is defined in a grammar by concatenating 2 strings that are made from character types, with a period in between.

An example of parameters in a list follows:

```
Grammar: cccccc.ddddd
Input String: WARNING.12225
```

- A wildcard list is defined in a grammar by enclosing a character type string in parentheses, followed by an asterisk. A wildcard list accepts an input list consisting of any number of strings, each of which matches the character type string, separated by periods (.).

An example of a wildcard list follows:

```
Grammar: (cccc)*
Input String: PARMA.PARMB.PARMC
```

The components of a list can be arbitrarily long when the **+** or **\*** matching character is coded in a list grammar. You can restrict the total length of a list to a maximum length by coding a colon (:) followed by the maximum length at the end of the grammar for the list parameter. For example, the following grammar:

```
a*.a*:20
```

accepts an input string consisting of two alphanumeric strings separated by a period. Each alphanumeric string may be of any length as long as the total length of the input list does not exceed 20 characters.

Similarly, the following grammar:

```
(a*)*:42
```

accepts an input string consisting of any number of alphanumeric strings separated by periods, as long as the total length of the input list does not exceed 42 characters.

## Positional Parameters

A positional parameter is a parameter that must be entered in a specific position of the input string syntax and before the keyword parameters.

The following are examples of positional parameters:

1) Grammar: ZDSMG ACTION SDA

Input string: ZDSMG ACTION SDA

- ZDSMG is a positional parameter and must always be the first parameter.
- ACTION is a positional parameter and must always be the second parameter.
- SDA is positional parameter and must always be the third parameter.

2) Grammar: P1 cc.dd (xx)\*

Input string: P1 AB.01 F0.E1.D2

- P1 is a positional parameter and must always be the first parameter.
- cc.dd is a positional parameter and must always be the second parameter.
- (xx)\* is positional parameter and must always be the third parameter.

A positional parameter can:

### 1. Match specific characters

Examples:

1) Grammar: ABC

Input string must be: ABC

2) Grammar: Abc

Input string can be: A or AB or ABC

### 2. Match character types: a, c, d, u, w, x, + and \*

Examples:

1) Grammar: acd\*

Input string can be: AX1 or 0Y12 or ZZ123 and so on

2) Grammar: d+++

Input string can be one to four digits: 1 or 21 or 345 or 5555 and so on

### 3. Be a list.

Example:

Grammar: acd.xw

Input string can be:

For the whole list: 0A1.F\*

For the first item: AB9.

For the second item: .0A

### 4. Be a wildcard list.

Example:

Grammar: (cd)\*

Input string: A1, or F2.C3, or D4.E5.Z2, or any combination of cd repeated up to 20 times. Each item in the input string must be delimited from the following item by a period (.).

## Keyword Parameters

A keyword parameter is a parameter whose value is determined by having a value assigned to the keyword name.

Keyword parameters must be entered after all the positional parameters are entered (if there are any positional parameters to be entered), and then the keyword parameters can be entered in any order. The following is an example of keyword and positional parameters:

ZSIPC ALTER INTERVAL TIME-xx PRIM=ccc ALTERN-xx

- ZSIPC is positional and must always be the first parameter.
- ALTER is positional and must always be the second parameter.

- INTERVAL is positional and must always be the third parameter.
- TIME-xx is a keyword and can be entered only in the fourth, fifth, or sixth positions.
- PRIM=ccc is a keyword and can be entered only in the fourth, fifth, or sixth positions.
- ALTERN-xx is a keyword and can be entered only in the fourth, fifth, or sixth positions.

There are two forms of keywords: regular and self-defining. Their grammar form and their range of possible values is different. Regular keywords have a range of many values, whereas self-defining keywords only have 2 possible values.

**Regular Keywords:** Regular keywords are keywords that can have a range of values.

- They have the following form:

```
Keyword name=keyword value
or
Keyword name-keyword value
```

The hyphen (-) and the equal sign (=) are keyword delimiters. No blanks are allowed between the keyword name, the delimiter (- or =), and the keyword value.

The keyword name can use only matching characters. The keyword value can use only character types, and lists (but not wildcard lists).

- Examples

The following are examples of valid keyword parameters:

```
KEY-aaaa
KEY=aaaaaa
```

The following are examples of keyword parameters that are not valid:

```
KEY - xxxxxx    blank between KEY and hyphen; and blank between
                  hyphen and xxxxxx
KEY -xxxxxx     blank between KEY and hyphen
KEY= xxxxxx     blank between equal sign and xxxxxx
KEY--xxxxxx     multiple occurrence of hyphen.
```

A regular keyword parameter can:

1. Match specific characters

The keyword name must match specific characters.

Example:

Grammar is: Key=cd.x

Input string for keyword name portion only:

```
K
KE
KEY
```

2. Match character types: a, c, d, u, w, x, + and \*

The keyword value can use character types.

Examples:

```
1) Grammar is: Key=cd
   Keyword value: cd
```

Input string for cd can be:

A1  
M1  
B5  
and so on...

2) Grammar is: KEY=d+++  
Keyword value: d+++

Input string for d+++  
can be 1 to 4 digits: 1 or 21, or 345, or 5555 and so on.

3. Be a list.

Only the keyword value can be a list.

Example:

Grammar is: KEYword=acd.xw  
Keyword value is a list: acd.xw

Input string can be:  
For the whole list: 0A1.F\*  
For the first item: AB9.  
For the second item: .0A

**Self-defining Keywords:** Self-defining keywords are keywords that return 1 of only 2 values:

- Y - a yes value
- N - a no value.

Self-defining keywords have the following form:

(NO)Keyword

- The input string KEYWORD has a value of Y (yes).
- The input string NOKEYWORD has a value of N (no).

Use matching characters for self-defining keywords.

Example:

- 1) Grammar is: (NO)WAlk  
Input can be:  
NOWALK - gives a value of N.  
WALK - gives a value of Y.  
NOWA - gives a value of N.  
and so on....
- 2) Grammar: (NO)ERASE  
Possible input strings:  
ERASE - the value assigned is Y.  
NOERASE - the value assigned is N.

## Translating Input String Values to Upper Case

When the IPRSE\_parse function is called with the IPRSE\_MIXED\_CASE option, the parser accepts input values in lower case. The values returned in the parser results can be translated to upper case by adding a less-than sign (<) at the end of each grammar token for which the matching value should be translated; otherwise the values returned in the parser results are the same as in the input string. In the following example, assume that the IPRSE\_MIXED\_CASE option is in effect.

Grammar: c c< c  
Input: x y z  
Result values: x Y z

## Programming Considerations for Grammars

Do not write ambiguous grammars, such as:

- (NO)NOnono
- [cccd] [XYZ0]
- K=w K=d
- ABcd ABxy Abnop

## Examples of Grammar Definitions

1. Grammar: `cc++`

Accepts any character string between 2 and 4 letters. If the IPRSE\_NOMIXED\_CASE option is used the input string must contain all uppercase letters.

Possible input strings: ABCD or ABC or AB and so on.

2. Grammar: `ccd+`

Accepts 2 letters followed by 1 or 2 decimal digits. If the IPRSE\_NOMIXED\_CASE option is used both letters must be uppercase.

Possible input strings: XY23 or XY2 and so on.

3. Grammar: `cc*`

Accepts any character string with at least 2 letters. If the IPRSE\_NOMIXED\_CASE option is used the input string must contain all uppercase letters.

Possible Input Strings: AB, or ABCD, or ABCDEFGHILLLO

4. Grammar: `cc.dd`

Accepts the following input strings:

- a. Two letters followed by 2 decimal digits:

AA.12  
XY.66

- b. A period followed by 2 decimal digits:

.45

- c. Two letters followed by a period:

AB.

If the IPRSE\_NOMIXED\_CASE option is used the input string must contain all uppercase letters.

5. Grammar: `(cc)*`

Accepts 2 letters or any number of multiples of 2 letters with periods between each 2 letters.

Input string: XX.XX.AB.CD or LA or DO.RE.MI and so on.

If the IPRSE\_NOMIXED\_CASE option is used the input string must contain all uppercase letters.

6. Example

Grammar `www`

Accepts 3 characters, including letters, decimal digits, underscores (`_`), and asterisks (`*`). If the IPRSE\_NOMIXED\_CASE option is used all of the letters must be in uppercase. An asterisk can match one or more w characters.

Input Strings:

\*X or ABC or \*1B

\* or \*\* or \*\*\*

A\* or \*B\* or 1\*2 or 0\*\* and so on.

## IPRSE\_parse—Parse a Text String against a Grammar

IPRSE\_parse matches an input string against an input grammar and produces a structure containing elements of the grammar with the corresponding elements of the input string. It is intended primarily for parsing TPF commands by real-time segments written in C.

IPRSE\_parse returns the parsed parameters and values through a pointer to a struct IPRSE\_output, declared in the tpfparse.h header.

### Format

```
#include <tpfparse.h>
int IPRSE_parse(char *string, const char *grammar,
                struct IPRSE_output *result, int options,
                const char *errheader);
```

#### string

The input string, which must be a standard C string terminated by a zero byte ('\0') or by an EOM character if the IPRSE\_EOM option is specified. If the IPRSE\_EOM option is specified, the maximum length of the string is 4095 characters.

#### grammar

The grammar describing acceptable input strings. The grammar must end in a zero byte ('\0').

#### result

The tokenized parameter list in the following form:

##### result.IPRSE\_parameter

The parameter name as specified by the grammar

##### result.IPRSE\_value

The value of the parameter as specified by the input string.

**Note:** This value is translated to upper case when the IPRSE\_MIXED\_CASE option is specified and the corresponding grammar parameter ends with a less-than sign (<).

##### result.IPRSE\_next

The pointer to the next entry in the output parameter list.

#### options

The following options control sending error messages:

##### IPRSE\_PRINT

Print all error messages.

##### IPRSE\_NOPRINT

Suppress all error messages. This is the default if IPRSE\_PRINT is not specified.

The following options control allocation of storage for the results:

##### IPRSE\_ALLOC

Obtain storage dynamically for the output structure. Always code this option.

##### IPRSE\_NOALLOC

This is the default if IPRSE\_ALLOC is not specified. Do not use IPRSE\_NOALLOC except to facilitate migration of old code that uses the IPRSE\_bldprstr function to initialize preallocated storage. Using the IPRSE\_ALLOC option is both more efficient and less likely to cause errors.

The following options are pertinent to input message blocks:

**IPRSE\_EOM**

If the input string ends with an EOM character (+) instead of an EOS character ('\0'), the IPRSE\_parse function replaces the EOM character with an EOS character. The first EOM character in the input string is replaced, there cannot be any '+' characters within the input string if the IPRSE\_EOM option is specified. The EOM character must be in the first 4095 characters of the input string.

**IPRSE\_NOEOM**

This is the default if IPRSE\_EOM is not specified. The input string must end with an EOS character ('\0').

The following four options control how the IPRSE\_parse function parses the input string. All four of these options can also be specified at the beginning of the grammar (see "Specifying Parser Options in the Grammar" on page 1449). Options specified in the grammar parameter override options specified in the options parameter:

**IPRSE\_STRICT**

Accept only spaces as token separators, and only dashes (-) as separators between keywords and values in the input string. Use this option when the input parameters can contain commas (,), slashes (/), or equal signs (=).

**IPRSE\_NOSTRICT**

Accept spaces, commas (,) or slashes (/) as token separators, and dashes (-) or equal signs (=) as separators between keywords and values. This is the default if IPRSE\_STRICT is not specified.

**IPRSE\_MIXED\_CASE**

Accept lowercase or uppercase letters in the input string.

**IPRSE\_NOMIXED\_CASE**

Accept only uppercase letters in the input string. This is the default if IPRSE\_MIXED\_CASE is not specified.

Multiple options can be ORed together; for example, IPRSE\_ALLOC | IPRSE\_PRINT.

**errheader**

A string that identifies the program calling the parser. This string is printed out as part of the error message text if the IPRSE\_PRINT option is specified.

**Normal Return**

IPRSE\_parse returns the number of parameters that have been parsed and put in the result structure. For example, a return code of 3 would mean that 3 parameters were parsed from an input string that contained 3 parameters.

**Note:** The return code must be used to count the nodes when traversing the result.

**Error Return**

IPRSE\_parse detects errors in the input string and in the grammar.

- Error in Input String
  - Return Codes
    - 1 The input string is a question mark (?) or HELP (represented by symbolic IPRSE\_HELP).

## IPRSE\_parse

- 0 The input string does not meet the requirements of the grammar (represented by symbolic IPRSE\_BAD).

- Error Messages

The IPRSE\_parse function issues the following messages. The cccc represents the errheader parameter that is printed after the message header; it shows which function or program was calling IPRSE\_parse when the error occurred. All messages will be sent via the wtopc function without chaining.

Number	Text
--------	------

<b>PRSE0001E</b>	cccc - TOO MANY PARAMETERS ENTERED
------------------	------------------------------------

<b>PRSE0004E</b>	cccc - INVALID USE OF PERIOD
------------------	------------------------------

<b>PRSE0005E</b>	cccc - INVALID ALPHANUMERIC CHARACTER
------------------	---------------------------------------

<b>PRSE0006E</b>	cccc - INVALID DECIMAL CHARACTER
------------------	----------------------------------

<b>PRSE0007E</b>	cccc - INVALID CHARACTER
------------------	--------------------------

<b>PRSE0008E</b>	cccc - INVALID HEXADECIMAL CHARACTER
------------------	--------------------------------------

<b>PRSE0009E</b>	cccc - MANDATORY PARAMETER NOT GIVEN
------------------	--------------------------------------

If the system can determine the last parameter in error, the message indicates the last parameter in error by adding the PARAMETER IN ERROR IS text and the parameter value.

<b>PRSE0011E</b>	cccc - INVALID INPUT PARAMETER
------------------	--------------------------------

If the system can determine the last valid parameter, the message indicates the last valid parameter by adding text that states LAST VALID PARAMETER IS and the parameter value. If a keyword was found to be in error, text will be added that states ERROR IN KEYWORD and the keyword.

<b>PRSE0014E</b>	cccc - TOO MANY CHARACTERS ENTERED
------------------	------------------------------------

<b>PRSE0015E</b>	cccc - TOO FEW CHARACTERS ENTERED FOR PARAMETER
------------------	--

- Error in Grammar
  - 00006F system error messages are displayed in console or dump when the grammar syntax is in error.
- 0007B system error messages occur when the parser is unable to obtain needed heap storage.

## Programming Considerations

- Always code the IPRSE\_ALLOC option. IPRSE\_NOALLOC and the IPRSE\_bldprstr are supported only for code that was written before the IPRSE\_ALLOC option was available.
- For information on creating a grammar, see “Defining a Grammar” on page 1448.

## Examples

A series of examples follows, the first of which is a complete program for creating and parsing with a grammar. All of the other examples show a grammar, its input string, and the IPRSE\_output structure.

The number of parameters found is returned if the string complies with the grammar conventions. See “Defining a Grammar” on page 1448 for additional information.



The examples represent the results in the IPRSE\_output structure through the diagram shown in Figure 4.

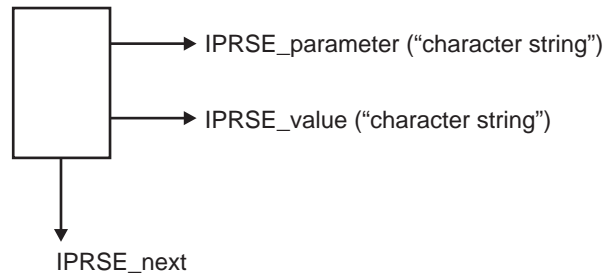


Figure 4. IPRSE\_output Structure

### Example 1: Coding Example for Grammar and Parser

The following example shows a program that:

- Parses input using a specific grammar (IPRSE\_parse)
- Uses the parsed output (process\_parm, defined in the example).

```

/*=====*/
/* This example shows a segment that parses */
/* a message in MIOMI format on data level D0. */
/* This code example includes calls to: */
/* - parse input using a specific grammar (IPRSE_parse) */
/* - use the parsed output (process_parm, defined in this segment) */
/*=====*/

#include <tpfeq.h>
#include <tpfapi.h>
#include <string.h>
#include <stdlib.h>
#include <tpfparse.h>

/*-----*/
/* Define the grammar for the command handled by this */
/* segment. */
/*-----*/

#define XMP_GRAMMAR "{ Positional " \
    " d+++ a.a [(xx)*] " \
    " (NO)SELFdef [Key-w List-cc.cc] " \
    "}"

/*-----*/
/* Declare an interface to functions that will process the parsed */
/* command parameters. */
/*-----*/

enum parm1_type { POSITIONAL_NOT_SPECIFIED, POSITIONAL_SPECIFIED };
enum parm5_type { SELFDEF_NOT_SPECIFIED, SELFDEF_NO, SELFDEF_YES };

struct xmp_interface
{
    enum parm1_type parm1_value; /* "Positional" */
    int parm2_value; /* "d+++ " */
    char *parm3_first; /* first "a" */
    char *parm3_second; /* second "a" */
    char *parm4_string; /* "(xx)*" */
    enum parm5_type parm5_value; /* "(NO)SELFdef" */
    char *parm6_value; /* "Key-w" */
    char *parm7_first; /* first "cc" */

```

## IPRSE\_parse

```
char          *parm7_second;    /* second "cc"          */
};

#define XMP_DEFAULTS { POSITIONAL_NOT_SPECIFIED, -1, NULL, NULL, NULL, \
                        SELFDEF_NOT_SPECIFIED, '\0', NULL, NULL }

/*-----*/
/* Declare internal function called by this segment.          */
/*-----*/

static void process_parm(struct xmp_interface *xi , char *p, char *v);

/*-----*/
/* Function ____ completes the parsing of the "Zxxxx" functional */
/* message contained in the core block on data level D0.          */
/*-----*/

void ____ (void)
{
    /*-----*/
    /* Define variables for accessing the command text in the */
    /* core block on D0.                                          */
    /*-----*/

    struct mi0mi      *block_ptr; /* pointer to core block */
    char              *input_ptr; /* pointer to message text */
    char              *eom_ptr;   /* pointer to _EOM character */
                                /* (to be replaced by '\0') */

    /*-----*/
    /* Define variables for the parser results.                  */
    /*-----*/

    struct IPRSE_output parse_results;
    int                  num_parms; /* For saving the IPRSE_parse */
                                /* return code.                */

    /*-----*/
    /* Define a moving pointer for traversing the parse results, a wtopc */
    /* header for the help message, and an interface variable for the */
    /* parsed parameter values.                                          */
    /*-----*/

    struct IPRSE_output *pr_ptr;
    struct wtopc_header msg_header;
    struct xmp_interface parm_values = XMP_DEFAULTS;

    /*-----*/
    /* Access the command block on level D0, point to the */
    /* beginning of the parameters by skipping over "Zxxxx", and replace */
    /* _EOM with '\0'.                                          */
    /*-----*/

    block_ptr = ecbptr()->celcr0;
    input_ptr = block_ptr->mi0acc + strlen("Zxxxx");
    eom_ptr = (char *)&block_ptr->mi0ln0 + block_ptr->mi0cct - 1;
    *eom_ptr = '\0';

    /*-----*/
    /* Call the parser.                                          */
    /*-----*/

    num_parms = IPRSE_parse(input_ptr, XMP_GRAMMAR, &parse_results,
                            IPRSE_ALLOC | IPRSE_PRINT, "TEST");
}
```

```

/*-----*/
/* Check if the command meets the grammar's requirements. */
/*-----*/

    if (num_parms > 0) /* The parse was successful; num_parms      */
                      /* parameters from the command            */
                      /* matched parameters specified in the      */
                      /* grammar (XMP_GRAMMAR).                   */
    {
        pr_ptr = &parse_results; /* point to the first result    */
        do
        {
            process_parm(&parm_values, pr_ptr->IPRSE_parameter,
                        pr_ptr->IPRSE_value);
            pr_ptr = pr_ptr->IPRSE_next;
        } while (--num_parms);

        /* call additional functions to further process the input */
    }
    else
    {
        if (num_parms == IPRSE_HELP)
        {
            wtopc_insert_header(&msg_header, "TEST", 99, 'I',
                                WTOPC_SYS_TIME);
            wtopc("EXAMPLE HELP MESSAGE", 0, WTOPC_NO_CHAIN,
                  &msg_header);
        }
        else ; /* IPRSE_parse has already written an error message. */
    }

    exit(0); /* Command processing is completed. */
}

/*****
/* Function process_parm sets the appropriate interface variable
/* field to the value corresponding to the matched parameter.
*****/

static void process_parm(struct xmp_interface *xi , char *p, char *v)
{

/*-----*/
/* Define the value that strcmp returns when the two strings passed
/* to it are equal.
/*-----*/

#define STRCMP_EQUAL 0

/*-----*/
/* Define a local variable to point to the dot in a list.
/*-----*/

    char *dot_ptr;

/*-----*/
/* Determine which parameter was matched and set up the appropriate
/* interface field(s) with the matching values.
/*-----*/

    if (strcmp(p, "Positional") == STRCMP_EQUAL)
    {
        xi->parm1_value = POSITIONAL_SPECIFIED;
    }

    else if (strcmp(p, "d++") == STRCMP_EQUAL)

```

## IPRSE\_parse

```
{
    xi->parm2_value = atoi(v);
}

else if (strcmp(p, "a.a") == STRCMP_EQUAL)
{
    dot_ptr = strchr(v, '.'); /* Point to the dot separating */
                             /* the two list sub-parameters. */
    *dot_ptr = '\0';          /* Divide the list parameter */
                             /* into two sub-strings. */
    xi->parm3_first = v;
    xi->parm3_second = dot_ptr + 1;
}

else if (strcmp(p, "xx") == STRCMP_EQUAL)
{
    xi->parm4_string = v;
}

else if (strcmp(p, "(NO)SELFdef") == STRCMP_EQUAL)
{
    xi->parm5_value =
        *v == 'Y' ? SELFDEF_YES : SELFDEF_NO;
}

else if (strcmp(p, "Key-w") == STRCMP_EQUAL)
{
    xi->parm6_value = *v;
}

else if (strcmp(p, "List-cc.cc") == STRCMP_EQUAL)
{
    dot_ptr = strchr(v, '.');
    *dot_ptr = '\0';
    xi->parm7_first = v;
    xi->parm7_second = dot_ptr + 1;
}

return;
}
```

If the grammar is:

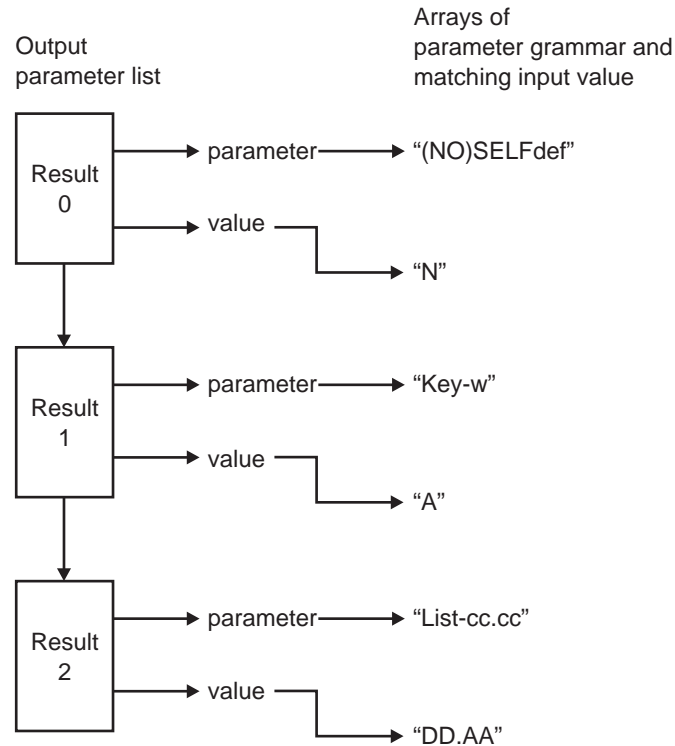
```
"{ Postional | d+++ a.a [(xx)*]
| (NO)SELFdef [Key-w List-cc.cc]}"
```

And the input string:

```
NOSELF K-A L-DD.AA
```

Therefore, the string meets the grammar requirements. IPRSE\_parse returns a return code of 3 because it parsed 3 parameters on the input string. The results in

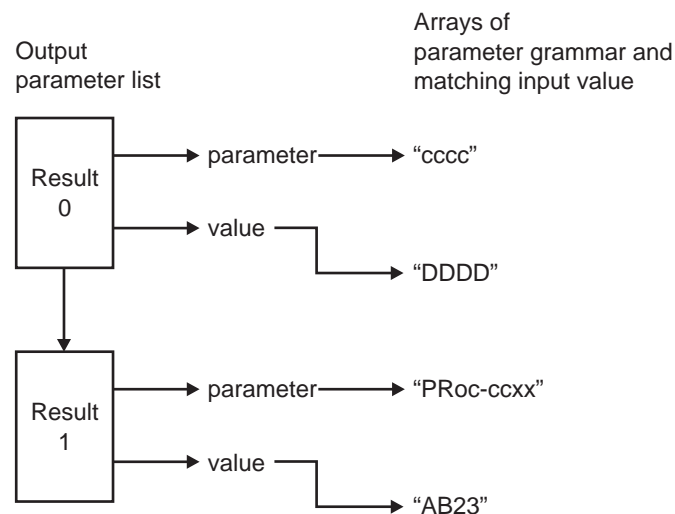
the arrays are shown as follows (result 0 is the first parameter parsed, and so on):



### Example 2

```
char *grammar = "cccc PRoc-ccxx [IS-d ]";
char *string = "DDDD PRO-AB23";
```

The string meets the grammar requirements. IPRSE\_parse returns a return code of 2 because it parsed 2 parameters from the input string. The results in the arrays are shown as follows (result 0 is the first parameter parsed, and so on):

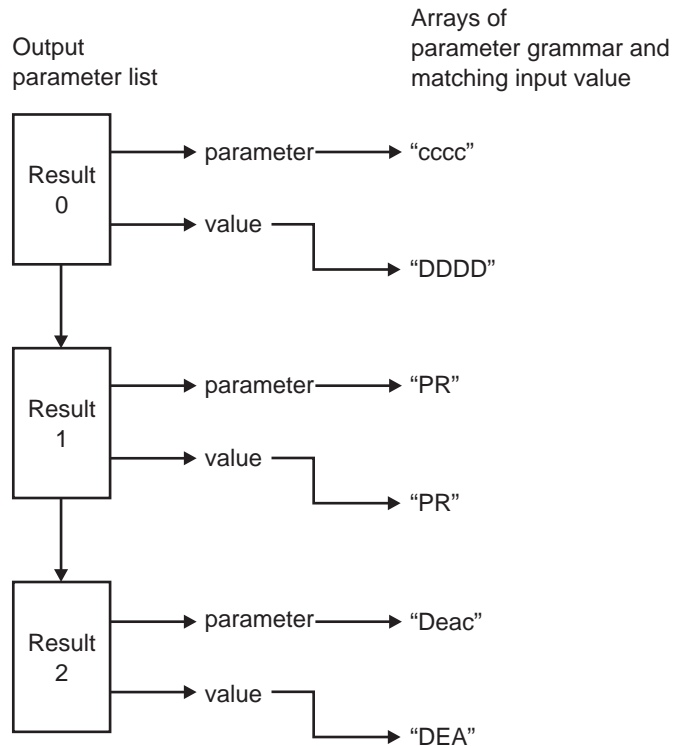


### Example 3

```
char *grammar = "cccc PR {Deac | Reac}";
char *string = "DDDD PR DEA ";
```

## IPRSE\_parse

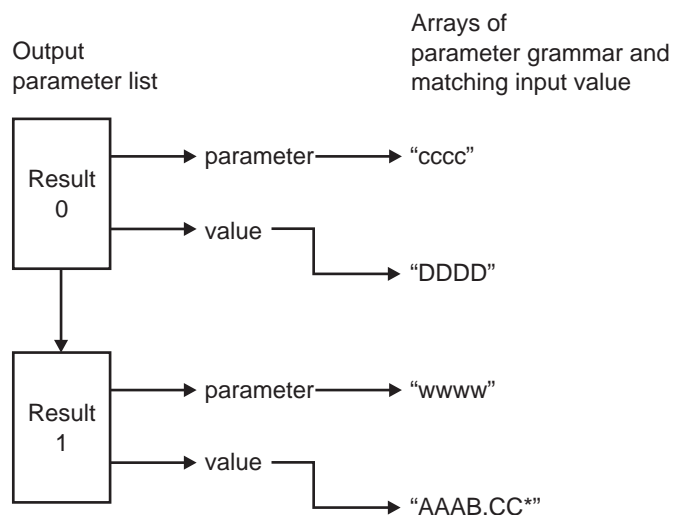
The string meets the grammar requirements. IPRSE\_parse returns a return code of 3 because it parsed 3 parameters from the input string. The results in the arrays are shown as follows (result 0 is the first parameter parsed, and so on):



### Example 4

```
char *grammar = "cccc (www)* ";  
char *string = "DDDD AAAB.CC* ";
```

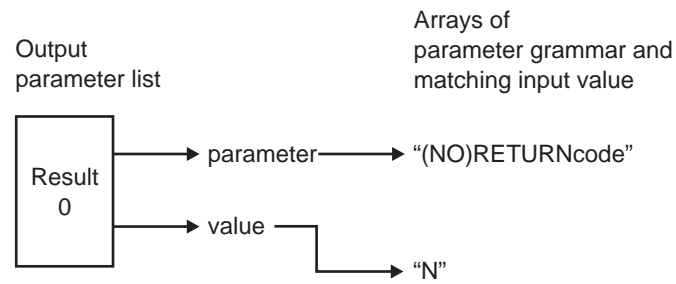
The string meets the grammar requirements. IPRSE\_parse returns a return code of 2 because it parsed 2 parameters from the input string. The results in the arrays are shown as follows (result 0 is the first parameter parsed, and so on):



### Example 5

```
char *grammar = "(NO)RETURNcode";  
char *string = "NORETURN";
```

The string meets the grammar requirements. IPRSE\_parse returns a return code of 1 because it parsed 2 parameters from the input string. The results in the arrays are shown as follows (result 0 is the first parameter parsed, and so on):



## Related Information

Macros BPKDC and BPPSC in *TPF General Macros*.

## IPRSE\_bldprstr—Initializing the Output Structure

### This function is obsolescent

Do not use the IPRSE\_bldprstr function. Instead code the IPRSE\_ALLOC option when calling the IPRSE\_parse function; this is both more efficient and less likely to result in errors.

The IPRSE\_bldprstr function was provided and required before TPF provided heap storage. The IPRSE\_bldprstr function should be used only in old code that was written before the IPRSE\_ALLOC option was available for the IPRSE\_parse function.

Function IPRSE\_bldprstr() is used to initialize an array of IPRSE\_output structures before passing it to IPRSE\_parse.

**Note:** Errors can occur if the size parameters of IPRSE\_bldprstr are too small. It is better to overestimate rather than underestimate size parameters.

## Format

```
#include <tpfparse.h>
void IPRSE_bldprstr (int parse_out_size,
                    struct IPRSE_output *parse_parm,
                    char *parm_ptr, int parm_size,
                    char *value_ptr, int value_size);
```

### parse\_out\_size

The maximum number of parameters that the grammar will accept at one time. (For example, the grammar {A|B|C} specifies 3 parameters, but there can be only 1 parameter in any given input string. Subsequently, parse\_out\_size would be 1.) Parse\_out\_size determines the length of the output parameter list, the grammar array, and the value array.

### parse\_parm,

A pointer to the output parameter list, which is an array of parse\_out\_size struct IPRSE\_outputs.

### parm\_ptr

A pointer to an array that will contain the parameters in the grammar. This array must be (parm\_size) multiplied by (parse\_out\_size) chars.

### parm\_size

The maximum length of a grammar parameter in the output parameter list. For example, to hold a grammar whose longest parameter is "TEST-c+++", parm\_size should be 11. The parameter contains a '\0' termination byte.

### value\_ptr

A pointer to an array that will contain the values of the parameters from the input string. This array must be (value\_size) multiplied by (parse\_out\_size) chars.

### value\_size

The length of the longest possible string needed to contain the value of the input parameter. For example, to hold a value of the input string whose longest possible input parameter from the grammar is "TEST-cccc", the value might be "ABBB" and then the value\_size should be 5. To determine the longest possible string, you must also count the terminating zero byte as part of the input value.



## Normal Return

Provides output structure; no special indicator provided.

## Error Return

None.

## Programming Considerations

- Do not use the IPRSE\_bldprstr function except in old code that cannot be updated to use the IPRSE\_ALLOC option. Instead use the IPRSE\_ALLOC option when calling the IPRSE\_parse function.
- Call IPRSE\_bldprstr before each call to IPRSE\_parse to initialize the output parameter list.
- The program takes an OPR dump if parse\_out\_size or value\_size is less than 1.
- It is recommended that you use the IPRSE\_alloc option for IPRSE\_parse.

## Examples

The following example is a subset of a program.

```
#define MAXPARAM 4          /* maximum number of parameters
                           that will be parsed          */

#define MAXPARAMLEN 15     /* length of the longest grammar
                           parameter (including '\0')    */

#define MAXVALULEN 15      /* length of the longest value
                           from input parameter (including '\0') */

/*-----*/
/* Defining parameter list and output arrays for grammar and
/* input values. In this case, they are automatically
/* allocated storage on the stack when the function begins
/* to run. (See also alternatives ways of getting
/* sections of storage in example notes.)
/*-----*/

struct IPRSE_output result_array [MAXPARAM];
char parm_array [MAXPARAM*MAXPARAMLEN];
char valu_array [MAXPARAM*MAXVALULEN];
IPRSE_bldprstr (MAXPARAM, result_array, parm_array, MAXPARAMLEN,
               value_array, MAXVALULEN);

/*-----*/
/* After call to IPRSE_bldprstr, call IPRSE_parse.
/*-----*/
```

**Note:** IPRSE\_bldprstr initializes the storage to X'00'.

## Related Information

“IPRSE\_parse—Parse a Text String against a Grammar” on page 1456.

**IPRSE\_bldprstr**

---

# Index

## Special Characters

- \_\_CREDC function 57
- \_\_CREEC function 59
- \_\_CREMC function 62
- \_\_CRETCL function 64
- \_\_CRETCL function 66
- \_\_CREXC function 69
- \_\_ENTDC function 104
- abort library function 7
- access library function 9
- addlc function 11
- alarm function 13
- assert macro 15
- atexit library function 17
- attac\_ext function 21
- attac\_id function 23
- attac function 19
- cebic\_goto\_bss function 25
- cebic\_goto\_dbi function 26
- cebic\_goto\_ssu function 27
- cebic\_restore function 28
- cebic\_save function 29
- chdir library function 30
- chmod library function 32
- chown library function 35
- cifrc function 37
- cinfc\_fast\_ss function 41
- cinfc\_fast function 40
- cinfc function 38
- clearerr library function 42
- close library function 44
- closedir library function 46
- closelog function 48
- cmaccp function 798
  - return codes 798
- cmallc function 800
  - return codes 800
- cmcfm function 803
  - return codes 803
- cmcfmd function 805
  - return codes 805
- cmdeal function 807
  - return codes 807
- cmecs function 810
  - return codes 810
- cmemn function 812
  - return codes 812
- cmepln function 814
  - return codes 814
- cmesl function 816
  - return codes 816
- cmflus function 818
  - return codes 818
- cmnit function 820
  - return codes 820
- cmptr function 823
  - return codes 823
- cmrcv function 825
  - return codes 826
- cmrts function 829
  - return codes 829
- cmsdt function 831
  - return codes 831, 833
- cmsed function 833
- cmsend function 835
  - return codes 835
- cmserr function 838
  - return codes 838
- cmsmn function 842
  - return codes 842
- cmspln function 844
  - return codes 844
- cmsptr function 846
  - return codes 846
- cmsrc function 848
  - return codes 848
- cmssl function 850
  - return codes 850
- cmsst function 852
  - return codes 852
- cmstpn function 854
  - return codes 854
- cmtrts function 856
  - return codes 856
- corhc function 49
- coruc function 50
- cratc function 51
- creat library function 54
- credc function 57
- creec function 59
- cremc function 62
- cretc\_level function 66
- cretc function 64
- crexc function 69
- crosc\_entrc function 71
- crusa function 73
- csonc function 75
- dbzac function 78
- dbzdc.h function 79
- defrc function 80
- deleteCache function 81
- deleteCacheEntry function 82
- deqc function 84
- detac\_ext function 86
- detac\_id function 88
- detac function 85
- dlayc function 90
- dllfree library function 91
- dllload library function 93
- dllqueryfn library function 94
- dllqueryvar library function 96
- dup library function 97
- dup2 library function 99
- ecbptr function 101
- enqc function 102

entdc macro	104
entrc macro	106
evinc function	108
evnqc function	109
evntc function	111
evnwc function	113
exit function	115
face_facs function	120
FACE function	117
FACS function	117
fchmod library function	123
fchown library function	125
fclose library function	127
fcntl library function	129
FD_CLR library function	134
FD_COPY library function	135
FD_ISSET library function	136
FD_SET library function	139
FD_ZERO library function	140
fdopen library function	137
feof library function	141
ferror library function	143
fflush library function	144
fgetc library function	146
fgetpos library function	148
fgets library function	150
file_record_ext function	161
file_record function	158
filec_ext function	154
filec function	152
fileno library function	156
filnc_ext function	167
filnc function	165
filuc_ext function	172
filuc function	170
find_record_ext function	181
find_record function	178
findc_ext function	176
findc function	174
finhc_ext function	187
finhc function	185
finwc_ext function	191
finwc function	189
fiwhc_ext function	195
fiwhc function	193
flipc function	197
flushCache function	198
fopen library function	199
fprintf library function	201
fputc library function	209
fputs library function	211
fread library function	213
freopen library function	215
fscanf library function	217
fseek library function	226
fsetpos library function	228
fstat library function	230
fsync library function	232
ftell library function	234
ftok function	235
ftruncate library function	237
fwrite library function	239
gdsnc function	241
gdsrc function	244
getc library function	246
getcc function	248
getchar library function	246
getcwd library function	252
getegid function	254
getenv function	255
geteuid function	256
getfc function	258
getgid function	260
getgrgid function	261
getgrnam function	263
getpc function	265
getpid function	267
getppid function	268
getpwnam function	269
getpwuid function	271
gets library function	273
gettimeofday library function	275
getuid function	276
glob_keypoint function	279
glob_lock function	280
glob_modify function	282
glob_sync function	284
glob_unlock function	286
glob_update function	288
glob function	277
global function	290
gsysc function	293
inqrc function	295
IPRSE_bldprstr function	1466
IPRSE_parse function	1456
keyrc_okey function	297
keyrc function	296
kill function	298
levtest function	300
link library function	302
lockc function	305
lodic_ext function	308
lodic function	306
longc function	312
longjmp library function	313
lseek library function	315
lstat library function	317
mail function	320
maskc function	325
mkdir library function	326
mkfifo function	328
mknod library function	331
MQBACK function	333
MQCLOSE function	335
MQCMIT function	337
MQCONN function	339
MQDISC function	341
MQGET function	343
MQINQ function	349
MQOPEN function	355
MQPUT function	359
MQPUT1 function	365

MQSET function 372  
 newCache function 376  
 numbc function 379  
 open library function 380  
 opendir library function 384  
 openlog function 386  
 pausc function 387  
 pause function 388  
 perror library function 389  
 pipe function 391  
 postc function 395  
 printf library function 201  
 progcc function 398  
 putc library function 400  
 putchar library function 400  
 puts library function 402  
 raisa function 404  
 raise library function 406  
 rcunc macro 408  
 read library function 410  
 readCacheEntry function 413  
 readdir library function 415  
 readlink library function 417  
 rehka function 419  
 relcc function 421  
 relfc function 423  
 relpc function 425  
 remove library function 427  
 rename library function 428  
 rewind library function 431  
 rewinddir library function 433  
 ridcc function 435  
 rlcha function 437  
 rmdir library function 439  
 routc function 441  
 rsysc function 443  
 rvtcc function 445  
 scanf library function 217, 447  
 selec function 448  
 select function 450  
 serrc\_op\_ext function 456  
 serrc\_op\_slc function 458  
 serrc\_op function 454  
 setbuf library function 460  
 setegid function 462  
 setenv function 463  
 seteuid function 466  
 setgid function 468  
 setjmp library function 470  
 setuid function 472  
 setvbuf library function 474  
 shmat function 476  
 shmctl function 478  
 shmdt function 481  
 shmget function 483  
 sigaction function 486  
 sigaddset macro 490  
 sigdelset macro 491  
 sigemptyset macro 492  
 sigfillset macro 493  
 sigismember macro 494

signal library function 495  
 sigpending function 498  
 sigprocmask function 499  
 sigsuspend function 502  
 sipcc function 504  
 sleep function 507  
 snapc function 508  
 sonic function 511  
 sprintf library function 201  
 sscanf library function 217  
 stat library function 513  
 strerror library function 517  
 swisc\_create function 518  
 symlink library function 521  
 systc function 526  
 system library function 527  
 tancc function 530  
 tape\_access function 532  
 tape\_close function 534  
 tape\_cntl function 535  
 tape\_open function 537  
 tape\_read function 538  
 tape\_write function 540  
 tasnc function 541  
 tbspc function 542  
 tcisc function 544  
 tdspc\_q function 546  
 tdspc\_v function 547  
 tdspc function 545  
 tdtac function 549  
 tmpfile library function 551  
 tmpnam library function 552  
 tmslc function 553  
 T02\_add function 883  
 T02\_addAllFrom function 886  
 T02\_addAtCursor function 1115  
 T02\_addAtIndex function 888  
 T02\_addKeyPath function 890  
 T02\_addRecoupIndexEntry function 893  
 T02\_allElementsDo function 1117  
 T02\_asOrderedCollection function 896  
 T02\_asSequenceCollection function 897  
 T02\_associateRecoupIndexWithPID function 899  
 T02\_asSortedCollection function 901  
 T02\_at function 903  
 T02\_atBrowseKey function 1224  
 T02\_atBrowseKeyPut function 1226  
 T02\_atBrowseNewKeyPut function 1228  
 T02\_atCursor function 1119  
 T02\_atCursorPut function 1121  
 T02\_atCursorWithBuffer function 1123  
 T02\_atDSdictKey function 1188  
 T02\_atDSdictKeyPut function 1190  
 T02\_atDSdictNewKeyPut function 1192  
 T02\_atDSsystemKey function 1194  
 T02\_atDSsystemKeyPut function 1196  
 T02\_atDSsystemNewKeyPut function 1198  
 T02\_atEnd function 1125  
 T02\_atKey function 905  
 T02\_atKeyPut function 907  
 T02\_atKeyWithBuffer function 909

T02_atLast function	1127	T02_createPIDinventoryKey function	1244
T02_atNewKeyPut function	911	T02_createReadWriteCursor function	1131
T02_atPut function	913	T02_createRecoupIndex function	998
T02_atRBA function	915	T02_createSequence function	1000
T02_atRBAPut function	917	T02_createSequenceTemp function	1002
T02_atRBAWithBuffer function	920	T02_createSequenceWithOptions function	1004
T02_atTPFKey function	1200	T02_createSet function	1006
T02_atTPFKeyPut function	1202	T02_createSetTemp function	1008
T02_atTPFNewKeyPut function	1204	T02_createSetWithOptions function	1010
T02_atTPFsystemKey function	1206	T02_createSort function	1012
T02_atTPFsystemKeyPut function	1208	T02_createSortedBag function	1013
T02_atTPFsystemNewKeyPut function	1210	T02_createSortedBagTemp function	1015
T02_atWithBuffer function	922	T02_createSortedBagWithOptions function	1017
T02_capture function	924	T02_createSortedSet function	1019
T02_changeDD function	1230	T02_createSortedSetTemp function	1021
T02_changeDS function	1232	T02_createSortedSetWithOptions function	1023
T02_class function	1346	T02_createSortTemp function	1026
T02_convertBinPIDtoEBCDIC function	1348	T02_createSortWithOptions function	1027
T02_convertClassName function	1234	T02_cursorMinus function	1133
T02_convertEBCDICtoBinPID function	1350	T02_cursorPlus function	1135
T02_convertMethodName function	1236	T02_defineBrowseNameForPID function	1246
T02_copyCollection function	927	T02_definePropertyForPID function	1028
T02_copyCollectionTemp function	929	T02_definePropertyWithModeForPID function	1031
T02_copyCollectionWithOptions function	931	T02_deleteAllPropertiesFromPID function	1034
T02_createArray function	933	T02_deleteBrowseName function	1248
T02_createArrayTemp function	935	T02_deleteCollection function	1036
T02_createArrayWithOptions function	937	T02_deleteCursor function	1137
T02_createBag function	939	T02_deleteDD function	1250
T02_createBagTemp function	941	T02_deleteDS function	1252
T02_createBagWithOptions function	943	T02_deleteEnv function	876
T02_createBLOB function	945	T02_deletePID function	1352
T02_createBLOBTemp function	947	T02_deletePropertyFromPID function	1038
T02_createBLOBWithOptions function	949	T02_deleteRecoupIndex function	1040
T02_createCursor function	1129	T02_deleteRecoupIndexEntry function	1041
T02_createDD function	1238	T02_first function	1139
T02_createDictionary function	951	T02_getAllPropertyNamesFromPID function	1043
T02_createDS function	1240	T02_getBLOB function	1045
T02_createDSwithOptions function	1242	T02_getBLOBWithBuffer function	1047
T02_createEnv function	874	T02_getBrowseDictPID function	1254
T02_createKeyBag function	954	T02_getClassAttributes function	1256
T02_createKeyBagTemp function	956	T02_getClassDocumentation function	1258
T02_createKeyBagWithOptions function	958	T02_getClassInfo function	1260
T02_createKeyedLog function	960	T02_getClassNames function	1262
T02_createKeyedLogTemp function	962	T02_getClassTree function	1264
T02_createKeyedLogWithOptions function	964	T02_getCollectionAccessMode function	1049
T02_createKeySet function	966	T02_getCollectionAttributes function	1266
T02_createKeySetTemp function	968	T02_getCollectionKeys function	1051
T02_createKeySetWithOptions function	970	T02_getCollectionName function	1268
T02_createKeySortedBag function	972	T02_getCollectionParts function	1270
T02_createKeySortedBagTemp function	974	T02_getCollectionType function	1053
T02_createKeySortedBagWithOptions function	976	T02_getCreateTime function	1272
T02_createKeySortedSet function	978	T02_getCurrentKey function	1141
T02_createKeySortedSetTemp function	980	T02_getCurrentKeyWithBuffer function	1143
T02_createKeySortedSetWithOptions function	982	T02_getDDAttributes function	1274
T02_createLog function	984	T02_getDirectoryForRRN function	1276
T02_createLogTemp function	986	T02_getDRprotect function	1055
T02_createLogWithOptions function	988	T02_getDSAttributes function	1278
T02_createOptionList function	990	T02_getDSdictPID function	1212
T02_createOrder function	995	T02_getDSnameForPID function	1280
T02_createOrderTemp function	996	T02_getErrorCode function	877
T02_createOrderWithOptions function	997	T02_getErrorText function	879



T02_getKeyPathAttributes function	1281	T02_removeRecoupIndexFromPID function	1086
T02_getListDDnames function	1283	T02_removeTPFKey function	1219
T02_getListDScollections function	1285	T02_removeTPFsystemKey function	1221
T02_getListDSnames function	1287	T02_removeValue function	1088
T02_getListUsers function	1289	T02_removeValueAll function	1090
T02_getMaxDataLength function	1057	T02_replaceBLOB function	1092
T02_getMaxKeyLength function	1058	T02_reset function	1178
T02_getMethodDocumentation function	1291	T02_restart function	1334
T02_getMethodNames function	1293	T02_restore function	1094
T02_getNumberOfRecords function	1295	T02_restoreAsTemp function	1096
T02_getPathInfoFor function	1296	T02_restoreWithOptions function	1098
T02_getPIDforBrowseName function	1299	T02_setClass function	1354
T02_getPIDinventoryEntry function	1301	T02_setCollectionAccessMode function	1101
T02_getPIDinventoryPID function	1303	T02_setDRprotect function	1103
T02_getPropertyValueFromPID function	1060	T02_setGetTextDump function	1335
T02_getRecordAttributes function	1305	T02_setKeyPath function	1180
T02_getRecoupIndex function	1307	T02_setMethodTrace function	1337
T02_getRecoupIndexForPID function	1310	T02_setPositionIndex function	1182
T02_getSortFieldValues function	1062	T02_setPositionValue function	1184
T02_getTPFDictPID function	1214	T02_setReadOnly function	1105
T02_getUserAttributes function	1312	T02_setSize function	1107
T02_includes function	1064	T02_size function	1109
T02_index function	1145	T02_taskDispatch function	1339
T02_isCollection function	1066	T02_validateCollection function	1340
T02_isDDdefined function	1314	T02_validateKeyPath function	1342
T02_isEmpty function	1147	T02_writeNewBLOB function	1111
T02_isExtended function	1316	topnc function	556
T02_isPropertyDefinedForPID function	1068	tourc function	557
T02_isTemp function	1070	toutc function	558
T02_key function	1149	tpcnc function	560
T02_keyWithBuffer function	1151	tpf_cfconc function	564
T02_last function	1153	tpf_cfdisc function	570
T02_locate function	1155	tpf_cresc function	572
T02_makeEmpty function	1072	tpf_decb_create function	576
T02_maxEntry function	1074	tpf_decb_locate function	578
T02_migrateCollection function	1318	tpf_decb_release function	580
T02_migrateDS function	1320	tpf_decb_swapblk function	582
T02_more function	1157	tpf_decb_validate function	584
T02_next function	1159	tpf_dlckc function	585
T02_nextPut function	1161	tpf_esfac library function	586
T02_nextRBAfor function	1163	tpf_fa4x4c function	592
T02_nextWithBuffer function	1165	tpf_fac8c function	587
T02_peek function	1168	tpf_faczc function	589
T02_peekWithBuffer function	1170	tpf_fork function	594
T02_previous function	1172	tpf_genlc function	599
T02_previousWithBuffer function	1174	tpf_gsvac function	602
T02_readOnly function	1076	tpf_help function	603
T02_reclaimPID function	1322	tpf_is_RPCServer_auto_restarted library function	607
T02_reconstructCollection function	1323	tpf_itrpc function	608
T02_recoupCollection function	1325	tpf_lemic function	612
T02_recoupDS function	1327	tpf_movec_EVM library function	618
T02_recoupPT function	1329	tpf_movec library function	616
T02_recreatedS function	1330	tpf_msg function	619
T02_remove function	1176	tpf_process_signals function	622
T02_removeBrowseKey function	1332	tpf_rcrfc function	624
T02_removeDSdictKey function	1215	tpf_RPC_options library function	626
T02_removeDSsystemKey function	1217	tpf_sawnc function	627
T02_removeIndex function	1078	tpf_select_bsd function	629
T02_removeKey function	1080	tpf_snmp_BER_encode function	632
T02_removeKeyPath function	1082	tpf_STCK function	634
T02_removeRBA function	1084	tpf_tcpip_message_cnt function	635

- tpf\_tm\_getToken function 636
- TPFxd\_archiveEnd function 1360
- TPFxd\_archiveStart function 1361
- TPFxd\_close function 1363
- TPFxd\_getPosition function 1365
- TPFxd\_getPrevPosition function 1367
- TPFxd\_getVOLSER function 1369
- TPFxd\_getVOLSERlist function 1371
- TPFxd\_nextVolume function 1372
- TPFxd\_open function 1374
- TPFxd\_read function 1376
- TPFxd\_readBlock function 1378
- TPFxd\_setPosition function 1380
- TPFxd\_sync function 1382
- TPFxd\_write function 1384
- TPFxd\_writeBlock function 1386
- tppc\_activate\_on\_confirmation function 737
  - return codes 738
- tppc\_activate\_on\_receipt function 741
  - return codes 741
- tppc\_allocate function 745
  - return codes 747
- tppc\_confirm function 750
  - return codes 750
- tppc\_confirmed function 753
  - return codes 753
- tppc\_deallocate function 755
  - return codes 756
- tppc\_flush function 759
  - return codes 759
- tppc\_get\_attributes function 761
  - return codes 762
- tppc\_get\_type function 763
  - return codes 763
- tppc\_post\_on\_receipt function 765
  - return codes 765
- tppc\_prepare\_to\_receive function 768
  - return codes 768
- tppc\_receive function 771
  - return codes 773
- tppc\_request\_to\_send function 777
  - return codes 777
- tppc\_send\_data function 779
  - return codes 780
- tppc\_send\_error function 782
  - return codes 783
- tppc\_test function 786
  - return codes 786
- tppc\_wait function 789
  - return codes 789
- tprdc function 641
- trewc function 643
- trsvc function 644
- tsync function 645
- twrtc function 646
- tx\_begin function 647
- tx\_commit function 649
- tx\_open function 651
- tx\_resume\_tpf function 652
- tx\_rollback function 654
- tx\_suspend\_tpf function 655

- uatbc function 657
- umask library function 659
- unfrc\_ext function 662
- unfrc function 661
- ungetc library function 664
- unhka function 666
- unlink library function 668
- unlkc function 670
- unsetenv function 671
- updateCacheEntry function 673
- utime library function 676
- vfprintf library function 678
- vprintf library function 680
- vsprintf library function 682
- wait function 684
- waitc function 686
- waitpid function 687
- WEXITSTATUS macro 690
- wgtac\_ext function 692
- wgtac function 691
- WIFEXITED macro 694
- WIFSIGNALED macro 695
- write library function 696
- WTERMSIG macro 699
- wtopc\_insert\_header function 704
- wtopc\_routing\_list function 706
- wtopc\_text function 707
- wtopc function 700
- xa\_commit function 708
- xa\_end function 710
- xa\_open function 712
- xa\_prepare function 714
- xa\_recover function 716
- xa\_rollback function 718
- xa\_start function 720

## A

- access and lock synchronizable TPF global field or record 280
- access mode 1409
- access mode, fopen 199
- access mode, retrieving for a collection 1049
- access mode, setting for a collection 1101
- access the group database by ID 261
- access the group database by name 263
- access the user database by name 269
- access the user database by user ID 271
- activating the archiving external device support 1361
- add
  - environment variable 463
- add a file descriptor to a file descriptor set 139
- add a new cache entry 673
- add a signal to a signal set 490
- adding
  - a key path to a collection 890
  - an item to the index 893
  - elements from the source collection to target collection 886
  - or replacing data to a BLOB 917
  - specified data to the collection 883



- address TPF global field or record 277
- address, getting for a method 1236
- addressing the ECB 101
- allocate shared memory 483
- API functions
  - See TPF API functions
- application dictionary, data store 1187
- assign general tape to ECB 541
- assigning buffers 460
- associate a stream 137
- associate an index with TPFCS 899
- attach a detached working storage block
  - attac\_ext 21
  - attac\_id 23
  - attac 19
- attach shared memory 476
- attach TPFAR database support structure 78
- attributes, getting
  - a data stores 1278
  - a users 1312
  - for a class 1256
  - for a collection 1266
  - of a record file address 1305

## B

- back out a queue 333
- backspace general tape and wait 542
- begin a global transaction 647
- BER encoding of SNMP variables 632
- binary
  - files 199
- browse support, APIs 1223
- browser dictionary
  - getting the PID 1254
  - removing the element from 1332
  - replacing the element from 1226
  - retrieving the element 1224
  - storing the element in 1228
- browser service functions
  - T02\_atBrowseKey 1224
  - T02\_atBrowseKeyPut 1226
  - T02\_atBrowseNewKeyPut 1228
  - T02\_changeDD 1230
  - T02\_changeDS 1232
  - T02\_convertClassName 1234
  - T02\_convertMethodName 1236
  - T02\_createDD 1238
  - T02\_createDS 1240
  - T02\_createDSwithOptions 1242
  - T02\_createPIDinventoryKey 1244
  - T02\_defineBrowseNameForPID 1246
  - T02\_deleteBrowseName 1248
  - T02\_deleteDD 1250
  - T02\_deleteDS 1252
  - T02\_getBrowseDictPID 1254
  - T02\_getClassAttributes 1256
  - T02\_getClassDocumentation 1258
  - T02\_getClassInfo 1260
  - T02\_getClassNames 1262
  - T02\_getClassTree 1264

## browser service functions *(continued)*

- T02\_getCollectionAttributes 1266
- T02\_getCollectionName 1268
- T02\_getCollectionParts 1270
- T02\_getCreateTime 1272
- T02\_getDDAttributes 1274
- T02\_getDirectoryForRRN 1276
- T02\_getDSAttributes 1278, 1337
- T02\_getDSdictPID 1212
- T02\_getDSnameForPID 1280
- T02\_getListDDnames 1283
- T02\_getListDScollections 1285
- T02\_getListDSnames 1287
- T02\_getListUsers 1289
- T02\_getMethodDocumentation 1291
- T02\_getMethodNames 1293
- T02\_getNumberOfRecords 1295
- T02\_getPIDforBrowseName 1299
- T02\_getPIDinventoryEntry 1301
- T02\_getPIDinventoryPID 1303
- T02\_getRecordAttributes 1305
- T02\_getTPFDictPID 1214
- T02\_getUserAttributes 1312
- T02\_isDDdefined 1314
- T02\_isExtended 1316
- T02\_isTemp 1070
- T02\_migrateCollection 1318
- T02\_migrateDS 1320
- T02\_reclaimPID 1322
- T02\_recoupCollection 1325
- T02\_recreatedDS 1330
- T02\_removeBrowseKey 1332
- T02\_restart 1334
- T02\_setGetTextDump 1335
- T02\_taskDispatch 1339
- T02\_validateCollection 1340
- buffer
  - assigning 460
  - flushing 144
  - format and print data 682
- buffer, data 859

## C

- C language function
  - See TPF API functions
- C language program
  - TPF file system functions 1405
- C parser utility, IPRSE 1447
- calculate file address 589
- call a program by name 562
- calling commands from a library function 527
- capture specified collection 924
- cause a key path to be validated 1342
- chained file records release 437
- change
  - data store definition 1232
  - environment variable 463
  - offset of a file 315
- change MDBF subsystem 25, 26
  - to BSS 25

- change MDBF subsystem *(continued)*
  - user 27
- change the mode of a file or directory 32
- changing a data definition 1230
- changing the attributes of a data store 1232
- character
  - reading with fgetc 146
  - reading with getc and getchar 246
  - ungetting 664
  - writing with fputc 209
  - writing with putc and putchar 400
- check available system resources 306, 308
- cipher program interface 37
- class name, getting a collections 1268
- cleanup operation, TPFxd\_close 1363
- clear the in-core pool reuse table 1329
- close a directory 46
- close a file 44
- close a general tape 534, 544
- close a queue 335
- close file 127
- close the system control log 48
- closing
  - files 127
  - logs 48
  - streams 127
- codes, error (summary table) 861, 874
- collection
  - defining a property for 1028, 1031
  - migrating a 1318
  - reconstruction 1323
  - retrieving a data store system 1285
  - returning property names for 1043
  - time stamp creation 1272
  - validation 1340
- collection access mode 1101
- collection access mode, retrieving 1049
- collection functions, TPFCS
  - T02\_add 883
  - T02\_addAllFrom 886
  - T02\_addAtIndex 888
  - T02\_addKeyPath 890
  - T02\_addRecoupIndexEntry 893
  - T02\_asSequenceCollection 897
  - T02\_associateRecoupIndexWithPID 899
  - T02\_asSortedCollection 901
  - T02\_at 903
  - T02\_atKey 905
  - T02\_atKeyPut 907
  - T02\_atKeyWithBuffer 909
  - T02\_atNewKeyPut 911
  - T02\_atPut 913
  - T02\_atRBA 915
  - T02\_atRBAPut 917
  - T02\_atRBAWithBuffer 920
  - T02\_atWithBuffer 922
  - T02\_capture 924
  - T02\_copyCollection 927
  - T02\_copyCollectionTemp 929
  - T02\_copyCollectionWithOptions 931
  - T02\_createArray 933

- collection functions, TPFCS *(continued)*
  - T02\_createArrayTemp 935
  - T02\_createArrayWithOptions 937
  - T02\_createBag 939
  - T02\_createBagTemp 941
  - T02\_createBagWithOptions 943
  - T02\_createBLOB 945
  - T02\_createBLOBTemp 947
  - T02\_createBLOBWithOptions 949
  - T02\_createKeyBag 954
  - T02\_createKeyBagTemp 956
  - T02\_createKeyBagWithOptions 958
  - T02\_createKeyedLog 960
  - T02\_createKeyedLogTemp 962
  - T02\_createKeyedLogWithOptions 964
  - T02\_createKeySet 966
  - T02\_createKeySetTemp 968
  - T02\_createKeySetWithOptions 970
  - T02\_createKeySortedBag 972
  - T02\_createKeySortedBagTemp 974
  - T02\_createKeySortedBagWithOptions 976
  - T02\_createKeySortedSet 978
  - T02\_createKeySortedSetTemp 980
  - T02\_createKeySortedSetWithOptions 982
  - T02\_createLog 984
  - T02\_createLogTemp 986
  - T02\_createLogWithOptions 988
  - T02\_createOptionList 990
  - T02\_createRecoupIndex 998
  - T02\_createSequence 1000
  - T02\_createSequenceTemp 1002
  - T02\_createSequenceWithOptions 1004
  - T02\_createSet 1006
  - T02\_createSetTemp 1008
  - T02\_createSetWithOptions 1010
  - T02\_createSortedBag 1013
  - T02\_createSortedBagTemp 1015
  - T02\_createSortedBagWithOptions 1017
  - T02\_createSortedSet 1019
  - T02\_createSortedSetTemp 1021
  - T02\_createSortedSetWithOptions 1023
  - T02\_definePropertyForPID 1028
  - T02\_definePropertyWithModeForPID 1031
  - T02\_deleteAllPropertiesFromPID 1034
  - T02\_deleteCollection 1036
  - T02\_deletePropertyFromPID 1038
  - T02\_deleteRecoupIndex 1040
  - T02\_deleteRecoupIndexEntry 1041
  - T02\_getAllPropertyNamesFromPID 1043
  - T02\_getBLOB 1045
  - T02\_getBLOBWithBuffer 1047
  - T02\_getCollectionAccessMode 1049
  - T02\_getCollectionKeys 1051
  - T02\_getCollectionType 1053
  - T02\_getCurrentKey 1141
  - T02\_getCurrentKeyWithBuffer 1143
  - T02\_getDRprotect 1055
  - T02\_getKeyPathAttributes 1281
  - T02\_getMaxDataLength 1057
  - T02\_getMaxKeyLength 1058
  - T02\_getPathInfoFor 1296

- collection functions, TPFCS (*continued*)
  - T02\_getPropertyValueFromPID 1060
  - T02\_getRecoupIndex 1307
  - T02\_getRecoupIndexForPID 1310
  - T02\_getSortFieldValues 1062
  - T02\_includes 1064
  - T02\_isCollection 1066
  - T02\_isPropertyDefinedForPID 1068
  - T02\_makeEmpty 1072
  - T02\_maxEntry 1074
  - T02\_reconstructCollection 1323
  - T02\_removeIndex 1078
  - T02\_removeKey 1080
  - T02\_removeKeyPath 1082
  - T02\_removeRBA 1084
  - T02\_removeRecoupIndexFromPID 1086
  - T02\_removeValue 1088
  - T02\_removeValueAll 1090
  - T02\_replaceBLOB 1092
  - T02\_restore 1094
  - T02\_restoreAsTemp 1096
  - T02\_restoreWithOptions 1098
  - T02\_setCollectionAccessMode 1101
  - T02\_setDRprotect 1103
  - T02\_setKeyPath 1180
  - T02\_setSize 1107
  - T02\_size 1109
  - T02\_validateKeyPath 1342
  - T02\_writeNewBLOB 1111
- collection summary table, cursors 1113, 1114
- collection support table, non-cursor APIs 881, 882
- commands, calling from library function 527
- commit a global transaction 649
- commit a queue 337
- commit work done for a transaction branch 708
- communications interface functions
  - inqrc 295
  - selec 448
- compiler functions supported as TPF extensions 1395
- compute low-level file address 117
- concurrent access 130
- conformance of TPF C support to ANSI/ISO standards 1389
- connect queue manager 339
- connect to a coupling facility (CF) cache structure 564
- connect to a coupling facility (CF) list structure 564
- consistent BLOB, creating 949
- control buffering 474
- control open file descriptors 129
- control program interface 38
- control system multiprocessor environment 387
- control tape 535
- convert an EVM address to an SVM address 602
- convert resource application interface 295
- convert system ordinal number 75
- converting
  - binary PID to an EBCDIC string 1348
  - EBCDIC class name to index 1234, 1262
  - EBCDIC string PID to a binary PID 1350
- converting file addresses 592
- copy documentation for a class 1258
- copy documentation of a method 1291
- copy the file descriptor set 135
- copying a creation time stamp 1272
- core level for occupied condition 300
- core lock program release 425
- coupling facility functions
  - deleteCache 81
  - deleteCacheEntry 82
  - flushCache 198
  - newCache 376
  - readCacheEntry 413
  - tpf\_cfconc 564
  - tpf\_cfdisc 570
  - updateCacheEntry 673
- CRAS status table 51
- create a child process 594
- create a new file or rewrite an existing file 54
- create a new logical record cache 376
- create an unnamed pipe 391
- create deferred entry
  - deferred entry 57
- create immediate entry 62
- create low-priority deferred entry 69
- create new ECB on specified I-stream 518
- create synchronous child ECBs 572
- create time-initiated entry 64, 66
- creating
  - adding or replacing data to a BLOB 917
  - BLOB 949
  - data definition 1238
  - data store definition 1240, 1242
  - empty array collection 937
  - empty persistent array collection 933
  - environment block 874
  - immediate entry 62
  - link to a file 302
  - locking cursor 1131
  - low-priority deferred entry 69
  - new ECB with an attached block 59
  - nonlocking cursor 1129
  - option list 990
  - PID associated index 998, 1040
  - PID inventory key 1244
  - symbolic link to path name 521
  - synchronous child ECB 572
  - temporary
    - bag 941, 982
    - BLOB 947
    - collection 897
    - copy 929
    - key bag 956
    - key set 968
    - key sorted bag 974, 976
    - key sorted set 980
    - keyed log 962
    - log 986
    - persistent identifier (PID) 935
    - sequence collection 1002
    - set 1008
    - sorted bag collection 901, 1015
    - sorted set collection 1021

- creating (*continued*)
  - temporary file 551, 552
  - time stamp 1272
  - time-initiated entry 64, 66
  - TPF collection support
    - bag 943
    - BLOB 945
    - copy and returning its PID 931
    - key bag 954, 958
    - key set 966, 970
    - key sorted bag 972
    - key sorted set 978
    - keyed log 960, 964
    - log 984, 988
    - sequence collection 1000, 1004
    - set collection 1006, 1010
    - sorted bag collection 1013
    - sorted set collection 1019
- cross-subsystem to enter a program and return 71
- current DD attribute values, getting 1274
- current directory for RRN, getting 1276
- current entry
  - delay processing 90
- current file position
  - effects of ungetc and ungetwc 226
- current file position, changing 226
- cursor API collection summary table 1113, 1114
- cursor functions, TPFCS
  - T02\_addAtCursor 1115
  - T02\_allElementsDo 1117
  - T02\_atCursor 1119
  - T02\_atCursorPut 1121
  - T02\_atCursorWithBuffer 1123
  - T02\_atEnd 1125
  - T02\_atLast 1127
  - T02\_createCursor 1129
  - T02\_createReadWriteCursor 1131
  - T02\_cursorMinus 1133
  - T02\_cursorPlus 1135
  - T02\_deleteCursor 1137
  - T02\_first 1139
  - T02\_index 1145
  - T02\_isEmpty 1147
  - T02\_key 1149
  - T02\_keyWithBuffer 1151
  - T02\_last 1153
  - T02\_locate 1155
  - T02\_more 1157
  - T02\_next 1159
  - T02\_nextPut 1161
  - T02\_nextRBAfor 1163
  - T02\_nextWithBuffer 1165
  - T02\_peek 1168
  - T02\_peekWithBuffer 1170
  - T02\_previous 1172
  - T02\_previousWithBuffer 1174
  - T02\_remove 1176
  - T02\_reset 1178
  - T02\_setPositionIndex 1182
  - T02\_setPositionValue 1184
- information about 1185

- cursor support 1113, 1185
- cycle up, restarting TPFCS 1334

## D

- data definition
  - changing 1230
  - creating 1238
  - testing to see if it is defined 1314
- data event control block (DECB)
  - converting file addresses 592
  - creating 576
  - locating 578
  - releasing 580
  - releasing a core block and file address 624
  - returning an 8-byte file address base 587
  - swapping a storage block 582
  - validating 584
- data event control block (DECB), creating 576
- data items
  - reading 213
  - writing 239
- data levels, interchanging status 197
- data list generation 599
- data store
  - application dictionary 1187
  - changing a definition 1232
  - creating a definition 1240, 1242
  - deleting 1252
  - get the dictionary PID 1212
  - migrating 1320
  - re-creating 1330
- data store system collections, retrieving 1285
- data stores, migrating 1320
- data stores, re-creating 1330
- DBI ID
  - restore previously saved 28
  - save current 29
- deadlocks 132
- decrement cursor to point to previous element 1133
- defer processing of current entry 80
- define
  - dequeue resource 84
  - enqueue resource 102
  - internal event 111
  - resource 49
- define and hold a resource 49
- defined data definition names, retrieving 1283
- defined users, getting a list of 1289
- defining a property for a collection 1028, 1031
- delay processing of current entry 90
- delete
  - environment variable 671
- delete a cache entry 82
- delete a file 427
- delete a logical record cache 81
- delete a signal from a signal set 491
- deleting
  - all defined properties 1034
  - an index 1086
  - an item from the index 1041

- deleting (*continued*)
  - collections 1036
  - cursor 1137
  - data definition 1250
  - data store 1252
  - environment block 876
  - persistent identifier (PID) and backing its store 1352
  - property 1038
- detach a working storage block from ECB 85, 86
- detach shared memory 481
- detach TPFAR database support structure 79
- detach working storage block from ECB 88
- determine signal 699
- determining a data store name for a PID 1280
- determining if a collection is persistent or temporary 1070
- device driver interface 1416, 1419, 1420, 1422, 1425, 1427, 1429, 1432, 1434, 1436
- device ID
  - lstat 317
  - stat 513
- dictionary
  - class 1187
  - symbol 1187
- dictionary support 1188
- dictionary support functions
  - T02\_atDSdictKey 1188
  - T02\_atDSdictKeyPut 1190
  - T02\_atDSdictNewKeyPut 1192
  - T02\_atDSsystemKey 1194
  - T02\_atDSsystemKeyPut 1196
  - T02\_atDSsystemNewKeyPut 1198
  - T02\_atTPFKey 1200
  - T02\_atTPFKeyPut 1202
  - T02\_atTPFNewKeyPut 1204
  - T02\_atTPFsystemKey 1206
  - T02\_atTPFsystemKeyPut 1208
  - T02\_atTPFsystemNewKeyPut 1210
  - T02\_removeDSdictKey 1215
  - T02\_removeDSsystemKey 1217
  - T02\_removeTPFKey 1219
  - T02\_removeTPFsystemKey 1221
- direct access storage 232
- directory
  - functions 1405
- dirty-reader protection status 1055, 1103
- disconnect from a coupling facility (CF) cache structure 570
- disconnect from a coupling facility (CF) list structure 570
- disconnect queue manager 341
- dispatching a TPFCS task 1339
- display
  - tape queue length 546
  - tape status 545
- DLL
  - explicit use 91, 93
  - freeing 91
  - loading 93
  - obtaining function pointers 94
  - obtaining variable pointers 96

- documentation, copy methods 1291
- drop previous programs and enter a program 104
- DS name, determining for a PID 1280
- duplicate 97
  - open file descriptor 97
  - open file descriptor to another 99

## E

- e-mail, processing 320
- EBW work area 518, 594
- EBW000 518, 595
- ECB control functions
  - defrc 80
  - dlayc 90
  - entdc 104
  - wait 684
  - waitc 686
  - waitpid 687
  - entrc 106
  - TPF\_CALL\_BY\_NAME 562
- ECB create functions
  - \_\_CREDC 57
  - \_\_CREEC 59
  - \_\_CREMC 62
  - \_\_CRETCL 64
  - \_\_CREXCL 66
  - \_\_CREXC 69
  - credc 57
  - creec 59
  - cremc 62
  - cretc\_level 66
  - cretc 64
  - crexc 69
  - swisc\_create 518
  - tpf\_cresc 572
  - tpf\_fork 594
- ECB exit 115
- ECB loading functions
  - pausc 387
- ECB reference 101
- ECB time slice facility 553
- electronic mail (e-mail), processing 320
- element
  - incrementing cursor to next 1135
  - iterating over all 1117
  - removing the element represented by the key 1219, 1221
  - retrieving using the specified key 1200, 1206
- empty persistent array collection, creating 933, 937
- encode SNMP variables in BER format 632
- end of field
  - clearing 431
  - flag 141
- end work performed for a transaction branch 710
- enter processing functions
  - entdc 104
  - entrc 106
  - TPF\_CALL\_BY\_NAME 562
- enter program and drop previous programs 104
- enter program with expected return 106, 562



- enterprise-specific traps for SNMP applications, sending 608
- environment collection
  - creating 874
  - deleting 876
- environment support functions
  - T02\_createEnv 874
  - T02\_deleteEnv 876
  - T02\_getErrorCode 877
  - T02\_getErrorText 879
- environment table
  - getenv function 255
  - setenv function 463
  - unsetenv function 671
- environment variable
  - add 463
  - change 463
  - delete 671
- environment variables
  - getenv function 255
  - setenv function 463
  - unsetenv function 671
- error
  - in files 143
  - indicator 143
  - indicator, clearing 431
- error code summary table 861, 874
- error handling, TPFCS 860
- error processing functions
  - perror 389
  - serrc\_op\_ext 456
  - serrc\_op\_slr 458
  - serrc\_op 454
  - snpc 508
- event
  - define internal 111
  - increment count 108
  - mark completion 113
  - query status 109
  - wait for completion with signal awareness 627
- event element completion 395
- examine and change blocked signals 499
- examine and change signal action 486
- examine pending signals 498
- execute a command 527
- exit an ECB 115
- EXIT\_FAILURE macro in stdlib.h 115
- EXIT\_SUCCESS macro in stdlib.h 115
- exiting a program 115
- extended options 154
- extended time-initiated entry 66
- external device interfaces
  - tape\_access 532
  - tdspc\_v 547
  - tdtac 549
  - tpcnc 560
  - TPFxd\_archiveEnd 1360
  - TPFxd\_archiveStart 1361
  - TPFxd\_close 1363
  - TPFxd\_getPosition 1365
  - TPFxd\_getPrevPosition 1367

## external device interfaces *(continued)*

- TPFxd\_getVOLSER 1369
- TPFxd\_getVOLSERlist 1371
- TPFxd\_nextVolume 1372
- TPFxd\_open 1374
- TPFxd\_read 1376
- TPFxd\_readBlock 1378
- TPFxd\_setPosition 1380
- TPFxd\_sync 1382
- TPFxd\_write 1384
- TPFxd\_writeBlock 1386
- error code table 1358, 1359

external device support 1357

## F

- FARF address conversion 75
- FARW initialization 120
- fast control program interface 40
- fast control program interface for any active subsystem 41
- FIFO special file, creating 328
- file
  - functions 1405
  - offset 696
  - writing to 696
- file a record
  - and unhold 170
  - and unhold with extended options 172
  - basic 152
  - higher level 158
  - with extended options: basic 154
  - with extended options: higher level 161
  - with no release 165
  - with no release and extended options 167
- file a record with extended options: basic 154
- file a record with extended options: higher level 161
- file a record with no release 165
- file a record with no release and extended options 167
- file a record: basic 152
- file a record: higher level 158
- file address
  - generation 120
  - low-level compute 117
- file address calculation 589
- file address generation 120
- file addresses, returning head of chain 1325
- file and unhold a record 170
- file and unhold a record with extended options 172
- file descriptor 1409
- file mode 32
  - chmod 32
- file permissions 9
  - access 9
- file pool storage release 423
- files
  - changing mode 123
  - descriptor 380, 391
  - descriptor flags 129
  - locking 130, 132
  - offset 380, 410

## files (continued)

- opening 199
- positioning 148, 226, 228, 234, 431
- renaming 428
- status flags 129

find

- a file record and wait 189
- a file record and wait with extended options 191
- a record 174, 178
- a record with extended options 176, 181
- and hold a file record 185
- and hold a file record and wait 193
- and hold a file record and wait with extended options 195
- and hold a file record with extended options 187

find a file record and wait 189

find a file record and wait with extended options 191

find a record 174, 178

find a record with extended options 176, 181

find and file functions

- file\_record\_ext 161
- file\_record 158
- filec\_ext 154
- filec 152
- filnc\_ext 167
- filnc 165
- filuc\_ext 172
- filuc 170
- find\_record\_ext 181
- find\_record 178
- findc\_ext 176
- findc 174
- finhc\_ext 187
- finhc 185
- finwc\_ext 191
- finwc 189
- fiwhc\_ext 195
- fiwhc 193
- rcunc 408
- unfrc\_ext 662
- unfrc 661

find and hold a file record 185

find and hold a file record and wait 193

find and hold a file record and wait with extended options 195

find and hold a file record with extended options 187

flags

- EOF 141
- file descriptor 129

flush the cache 198

flush the contents of the cache 198

format and print data

- to a stream 678
- to buffer 678
- to standard output stream 678

formatted I/O 201

free core storage block if held 73

function 523

functions

- path name rules 1408
- TPF file system 1405

## G

general data set functions

- gdsnc 241

general file get file address 404

general file support functions

- raisa 404

general tape

- tape\_cntl 535
- assign to ECB 541
- backspace and wait 542
- close 534, 544
- open 537, 556
- read 641
- read record from 538
- reserve 644
- rewind and wait 643
- synchronize 645
- write a record 646
- write a record to 540

generate a data list 599

generate a token 235

generate file address 120

get

- value of environmental variables 255

get a list of prepared transaction branches 716

get data set entry 241

get file pool address

- See getfc function

get general data set record 244

get input string 273

get message from an open queue 343

get pointer to run-time error 517

get program and lock in core 265

get status information 230

get storage from the system heap 293

get symbolic file address information 511, 586

get the effective group ID 254

get the effective user ID 256

get the file descriptor 156

get the real group ID 260

get the real user ID 276

get the unique token for the current transaction 636

get working storage block

- See getcc function

getting

- address of a method 1236
- an index 1307
- an index key 1310
- attributes
  - for a collection 1266
  - for a data store 1278
  - of a class 1256
  - of a record 1305
  - of a user 1312
  - of key paths 1281
- browser dictionary PID 1254
- class name of the collection 1268
- current data definition attribute values 1274
- current directory for RRN 1276
- current key 1141
- current key in the buffer 1143

## getting (continued)

- dictionary PID of a data store 1212
- file information 513
- file position, fgetpos 148
- information for a class 1260
- inheritance tree for a class 1264
- list of defined users 1289
- method names for a class 1293
- part names for a collection 1270
- path name of the working directory 252
- PID inventory entry 1301
- PID of PID inventory 1303
- read-only attribute of the collection 1076
- status of a file 317
- status of symbolic link 317
- TPF dictionary's PID 1214
- type value of the collection 1053
- user class ID of the collection 1346

global functions

- glob\_keypoint 279
- glob\_lock 280
- glob\_modify 282
- glob\_sync 284
- glob\_unlock 286
- glob\_update 288
- glob 277
- global 290

global support utility, GNTAGH 1437

global tag

- format 1441

global tagnames, C language

- creating on MVS 1439
- creating on VM 1438
- format 1441

global transaction 647

- tx\_begin function 647
- tx\_commit function 649
- tx\_resume\_tpf function 652
- tx\_suspend\_tpf function 655
- begin 647
- commit 649
- resume 652
- rollback 654
- suspend 655

globals

- converting assembler to C 1437

GNTAGH user's guide 1437

group ID 35

- chown 35
- lstat 317
- stat 513

## H

- head of chain, returning file addresses 1325
- help messages 603
- higher level file a record 158
- hold a resource 49

## I

- I-stream 518
- importing functions and variables 93
- in-core pool reuse table, clearing 1329
- increment count for event 108
- incrementing
  - cursor to next element 1135
  - returning the next element 1159, 1165
- information, getting for a class 1260
- initial setup for external device support 1357
- initialize a FARW 120
- initialize and empty a signal set 492
- initialize and fill a signal set 493
- initialize the file descriptor set 140
- inode
  - stat 317, 513
- input message tokenization 1447
- input/output
  - error testing 143
  - opening files 199
- inquire about object attributes 349
- insert specified data in the collection 888, 1115
- Install signal handler 495
- interchange the status of two data levels 197
- interface
  - cipher program 37
  - control program 38
  - fast control program 40
  - fast control program for any active subsystem 41
- internal format of collections 1316
- Internet mail, processing 320
- inventory key, creating 1244
- inventory, PID 1303
- IPC\_CREAT symbolic constant 483
- IPC\_EXCL symbolic constant 483
- IPC\_PRIVATE symbolic constant 483
- IPC\_RMID symbolic constant 478
- IPC\_SET symbolic constant 478
- IPC\_STAT symbolic constant 478
- IPRSE 1447
- ISO-C
  - TPF file system functions 1405
- issue
  - snapshot dump 508
  - system error extended: operational 456
  - system error SLIST: operational 458
  - system error with message 389
  - system error: operational 454
- issue a message 619
- issue a user specified control operation CCW 560
- issue a user specified data transfer CCW 549
- issue help messages 603
- iterate over all elements 1117

## K

- keypoint TPF global field or record 279
- keys
  - replacing the element 1190, 1196
  - storing the element 1192



kill a signal 298

## L

last element, pointing cursor at 1153

line

reading with fgets() 150

writing with puts() 402

link

functions 1405

link count 302

locate terminal entry 691

locate terminal entry with extended options 692

locating a data event control block (DECB) 578

locating the key and pointing the cursor at its element 1155

lock

and access synchronizable TPF global field or record 280

program in core 265

resource 305

lock entry management interface 585, 612

locking cursor 1131

log

closing 48

opening 386

sending a message to 523

long running ECB 553

low-level file address compute 117

low-priority deferred entry 69

## M

macros, TPF

C function related 3

major device number 331, 1412

make a directory 326

make a FIFO special file 328

mark event completion 113

mark event element completion 395

matching failure 222

maximum existence time setting 312

MDBF user attribute reference request 657

message

tpf\_msg 619

parsing 1447

message counter

tpf\_tcpip\_message\_cnt 635

message routing 441

method trace table, setting 1337

migrating

collections 1318

data stores 1320

minor device number 331, 1422

miscellaneous collection support

T02\_class 1346

T02\_convertBinPIDtoEBCDIC 1348

T02\_convertEBCDICtoBinPID 1350

T02\_deletePID 1352

T02\_readOnly 1076

T02\_setClass 1354

miscellaneous collection support (*continued*)

T02\_setReadOnly 1105

MMCCCHR address conversion 75

mode

changing 123, 125

mode of access 1409

mode, fopen 199

modify

PSW mask bits 325

TPF global field or record 282

modify program status word mask bits 325

monitor read, write, and exception status 450, 629

mount or positioning required, TPFxd\_nextVolume 1372

mount or positioning required, TPFxd\_open 1374

move

data between EVM and SVM 616

data from one EVM to another EVM 618

moving a VIPA to another processor 637

multiprocessor environment system control 387

multitasking functions

corhc 49

coruc 50

deqc 84

enqc 102

evinc 108

evnqc 109

evntc 111

evnwc 113

postc 395

tpf\_STCK 634

## N

named pipe, creating 328

names, returning property 1043

NDEBUG

access 9

non-cursor API collection summary table 881

nonlocal goto

longjmp 313

setjmp 470

nonlocking cursor, creating 1129

## O

obtain

file pool address 258

input string 273

symbolic file address information 511, 586

working storage block 248

obtain a process ID 267

obtain child exit status 690

obtain status information from a child process 684, 687

obtain the parent process ID 268

obtain value of environment variables

See getenv function

open

a directory 384

a file 380

a log 386

- open a general tape 537, 556
- open a queue 355
- open a resource manager 712
- open a set of resource managers 651
- open file description 1409
- open the system control log 386
- opening
  - files 199
  - logs 386
  - streams 199, 215
- opening the archive interface 1361
- operate on TPF global field 290
- option list, creating 990
- original protection key 297
- ownership of files/directories 125

## P

- parser 1447
- parsing functions
  - IPRSE\_bldprstr 1466
  - IPRSE\_parse 1456
- PAT slot address return 398
- permission 1409
  - functions 1405
  - handling in programs 1409
- persistent sorted bag collection, creating 1017
- persistent sorted set collection, creating 1023
- PID inventory entry, getting 1301
- PID, determining the DS name 1280
- PID, getting for the TPF dictionary 1214
- pointers 1409
- pointing
  - cursor at a specified element 1182, 1184
  - cursor at first element 1139
  - cursor at last element 1153
- pool management functions
  - getfc 258
  - relfc 423
  - rlcha 437
- POSIX library functions
  - abort 7
  - access 9
  - chdir 30
  - chmod 32
  - chown 35
  - clearerr 42
  - close 44
  - closedir 46
  - closelog 48
  - creat 54
  - dup 97
  - dup2 99
  - fchmod 123
  - fchown 125
  - fclose 127
  - fcntl 129
  - fdopen 137
  - feof 141
  - ferror 143
  - fflush 144

## POSIX library functions *(continued)*

- fgetc 146
- fgetpos 148
- fgets 150
- fileno 156
- fopen 199
- fprintf 201
- fputc 209
- fputs 211
- fread 213
- freopen 215
- fscanf 217
- fseek 226
- fsetpos 228
- fstat 230
- fsync 232
- ftell 234
- ftok 235
- ftruncate 237
- fwrite 239
- getc 246
- getchar 246
- getcwd 252
- getegid 254
- geteuid 256
- getgid 260
- getgrgid 261
- getgrnam 263
- getpid 267
- getppid 268
- getpwnam 269
- getpwuid 271
- gets 273
- getuid 276
- kill 298
- link 302
- lseek 315
- lstat 317
- mkdir 326
- mkfifo 328
- mknod 331
- open 380
- opendir 384
- openlog 386
- pause 388
- perror 389
- pipe 391
- printf 201
- putc 400
- putchar 400
- puts 402
- read 410
- readdir 415
- readlink 417
- remove 427
- rename 428
- rewind 431
- rewinddir 433
- rmdir 439
- scanf 217
- setbuf 460

## POSIX library functions *(continued)*

- setegid 462
- seteuid 466
- setgid 468
- setuid 472
- setvbuf 474
- shmat 476
- shmctl 478
- shmdt 481
- shmget 483
- sigaction 486
- sigaddset 490
- sigdelset 491
- sigemptyset 492
- sigfillset 493
- sigismember 494
- sigpending 498
- sigprocmask 499
- sigsuspend 502
- sleep 507
- sprintf 201
- sscanf 217
- stat 513
- symlink 521
- syslog 523
- system 527
- tmpfile 551
- tmpnam 552
- tpf\_dlckc 585
- tpf\_is\_RPCServer\_auto\_restarted 607
- tpf\_lemic 612
- tpf\_moved 616
- tpf\_process\_signals 622
- tpf\_RPC\_options 626
- umask 659
- ungetc 664
- unlink 668
- utime 676
- vfprintf 678
- vprintf 680
- vsprintf 682
- wait 684
- waitpid 687
- WEXITSTATUS 690
- WIFSIGNALED 695
- write 696
- WTTERMSIG 699
- precision argument, fprintf family 204
- prepare to commit 714
- preserve stack environment 470
- process outstanding signals 622
- processing electronic mail (e-mail) 320
- processing Internet mail 320
- program status word mask bits 325
- program termination
  - atexit library function 17
  - exit library function 115
  - a program 115
  - abnormal program termination
    - abort library function 7
    - assert function 15

- property
  - deleting 1034, 1038
  - names, returning 1043
- protection key 296, 297
- protection status, dirty-reader 1055, 1103
- pushing characters back to input stream 664
- put a message on an open queue 359
- put a single message on a queue 365
- put string to terminal 402

## Q

- query
  - event status 109
  - number of storage blocks available 379
  - RPC server restarted 607
- query child status 694, 695

## R

- raise 622
- raise condition 406
- raise outstanding signals 622
- random
  - access 226, 234
- re-creating
  - data stores 1330
- read
  - character from stdin 146, 246
  - character from stream 246
  - data items from stream 213
  - directory, readdir library function 415
  - formatted 217
  - from file, read library function 410
  - line from stream 150
  - read a string, fgets() library function 150
  - scanning 217
  - value of symbolic link, readlink library function 417
- read a cache entry 413
- read a record from general tape 538
- read data from an external device into a buffer,  
TPFxd\_read 1376
- read general tape record 641
- read into a core block from an external device,  
TPFxd\_readBlock 1378
- read operations with fgetc 146
- reading
  - a character 246
  - complete or partial BLOB 915, 920
- real-time tape
  - synchronize 645
  - write real-time tape record 558
  - write real-time tape record and release storage  
block 557
- reclaim the PID 1322
- reconstructing a collection 1323
- record file address, T02\_getRecordAttributes 1305
- records, returned by T02\_getNumberOfRecords 1295
- redirect
  - streams, using freopen 215
- rehook core block 419

- reinitializing and discarding elements in the collection 1072
- release
  - chained file records 437
  - file pool storage 423
  - program from core lock 425
  - storage from the system heap 443
  - working storage block 421
- release core block and unhold file record 408
- releasing a core block and file address 624
- releasing a data event control block (DECB) 580
- remove a directory 439
- remove a file descriptor 134
- removing
  - all elements with specified value from collection 1090
  - element from the browser dictionary 1332
  - element represented by the key 1219, 1221
  - element the cursor is pointing at 1176
  - element using the key 1215, 1217
  - first element with specified value from the collection 1088
  - key paths from a collection 1082
  - RBA 1084
  - specified element at index 1078
- removing directory entry
  - entry removal 668
  - opening 384
  - reading with readdir 415
  - removing 439
  - renaming 428
  - repositioning 433
  - rewinding 433
  - working 252
- renaming files 428
- reopening streams 215
- replacing
  - element from the browser dictionary 1226
  - element using the key 1190, 1196
  - element using the specified key 1202, 1208
  - specified entry's data with specified data 913
- request completion, informing the archive facility, TPFxd\_archiveEnd 1360
- reserve general tape 644
- reset error and end-of-file 42
- resetting cursor to point to the first element 1178
- residency of TPFCS collections 1316
- resource
  - define 49
  - hold 49
  - unhold 50
- resource application interface 295
- resource lock 305
- resource manager 651, 712
  - tx\_open function 651
  - open 651, 712
- resource vector table entry search 445
- restarting TPFCS at cycle up 1334
- restore
  - previously saved DBI and SSU IDs 28
  - protection key 296, 297
- restore stack environment 313
- restoring
  - a captured collection 1094
  - a collection as a temporary collection 1096
  - a collection using the specified options 1098
- resume a global transaction 652
- retrieving
  - attributes of key paths 1281
  - current directory for the specified RRN 1276
  - current key 1141
  - current key in the buffer 1143
  - current positioning information,
    - TPFxd\_getPosition 1365
  - current VOLSER and media type 1369
  - current VOLSER list and media type 1371
  - defined data definition names 1283
  - defined data store names 1287
  - element from the browser dictionary 1224
  - element using the specified key 1188, 1194, 1200, 1206
  - previous positioning information,
    - TPFxd\_getPosition 1367
  - property names for a collection 1043
  - value for a property 1060
  - volume serial (VOLSER) for a specified tape name 547
- retrieving a collection access mode 1049
- return a value for the file descriptor 136
- return PAT slot address 398
- return sequence collection 1327
- returning
  - associated error text 879
  - copy of the specified PID of the collection 927
  - current index position 1145
  - current key value 1149, 1151
  - data in specified position in the specified collection 903
  - data in the supplied buffer 922
  - element cursor points at 1119, 1123
  - error codes 877
  - head of chain file addresses 1325
  - maximum data length value 1057, 1058
  - maximum number of entries for the 1074
  - next element 1170
  - next element with no cursor movement 1168
  - number of records 1295
  - return to the next x bytes 1163
  - sort field values 1062
  - to the previous element 1172, 1174
- returning an 8-byte file address base 587
- rewind a stream 431
- rewind general tape and wait 643
- roll back work done for a transaction branch 718
- rollback a global transaction 654
- route a message 441

**S**

- S\_IRGRP symbolic constant 483
- S\_IROTH symbolic constant 484
- S\_IRUSR symbolic constant 483

- S\_ISBLK(mode) macro 317
- S\_ISCHR(mode) macro 317, 514
- S\_ISDIR(mode) macro 317, 514
- S\_ISLNK(mode) macro 317, 514
- S\_ISREG(mode) macro 317, 514
- S\_IWGRP symbolic constant 484
- S\_IWOTH symbolic constant 484
- S\_IWUSR symbolic constant 483
- save
  - header for wtopc 704
  - routing list for wtopc 706
- save current DBI and SSU IDs 29
- save stack environment 470
- scan input for variables 447
- schedule an alarm 13
- search CRAS status table 51
- searching
  - collection for a specified key 905
  - collection for an element 1064
  - CRAS status table 51
  - deleting the specified key from the collection 1080
  - RVT entries 445
  - specified collection for a specified key 907, 909, 911
- security
  - functions 1405
  - handling in programs 1409
- see if a signal is a member of a signal set 494
- select a thread application interface 448
- select or poll an open file descriptor 1425
- select or poll cleanup of an open file descriptor 1427
- send
  - enterprise-specific trap to SNMP application 608
  - system message 700, 707
- send a message to the control log 523
- send a signal to a process 298
- sequence collection, returning 1327
- serial number
  - lstat 317
  - stat 513
- set entry maximum existence time 312
- set object attributes 372
- set signal mask and wait for a signal 502
- set the effective group ID 462
- set the effective user ID 466
- set the group ID 468
- set the real user ID 472
- setting
  - access mode for a collection 1101
  - current position, TPFXd\_setPosition 1380
  - cursor to use a specific path 1180
  - file access and modification times 676
  - file mode creation mask 659
  - method trace on or off 1337
  - read-only attribute of the collection 1105
  - RPC options 626
  - size of a collection 1109
  - size of BLOB 1107
  - text dump on or off 1335
  - user class ID of the collection 1354
- setup for external device support 1357
- shared memory control 478
- shared memory functions
  - shmat 476
  - shmctl 478
  - shmdt 481
  - shmget 483
- sig argument in signal library function 495
- signal
  - handler 495
- SNA RID conversions 435
- snapshot dump 508
- SNMP application traps, sending 608
- SNMP variable encoding in BER format 632
- special file, create a 331
- special file, inheritance 1413
- specified PID of the collection, creating and returning 927
- SSU ID
  - restore previously saved 28
  - save current 29
- stack
  - restoring the environment 313
  - saving an environment 470
- standard C header files 1393
  - assert.h 1393
  - ctype.h 1393
  - errno.h 1393
  - float.h 1393
  - limits.h 1393
  - locale.h 1393
  - math.h 1393
  - stdarg.h 1393
  - stddef.h 1393
  - stdio.h 1393
  - stdlib.h 1393
  - string.h 1393
  - sys/ipc.h 1393
  - sys/shm.h 1393
  - time.h 1393
- standard C/C++ functions not documented 1397
- start work for a transaction branch 720
- stat structure 513
- stdout
  - format and print data 680
- stdout, format and print data 680
- storage blocks available 379
- storage management functions
  - attac\_ext 21
  - attac\_id 23
  - attac 19
  - crusa 73
  - detac\_ext 86
  - detac\_id 88
  - detac 85
  - getcc 248
  - rehka 419
  - relcc 421
  - unhka 666
- store clock 634
- storing
  - as the next element 1161

- storing (*continued*)
  - in the browser dictionary 1228
  - using the key 1192
  - using the new key 1198, 1204, 1210
  - where cursor points 1121
- stream
  - access mode 215
  - associating with file descriptor 137
  - binary mode 215
  - buffering 460
  - changing current file position 226, 234
  - changing file position 431
  - closing 127
  - EOF (end of file) 141
  - formatted I/O 201, 217
  - Input/Output 127
  - opening 199
  - reading characters with `fgetc` 146
  - reading characters with `getc` and `getchar` 246
  - reading data items with `fread()` 213
  - reading lines with `fgets()` 150
  - redirection 215
  - reopening 215
  - rewinding 431
  - text mode 215
  - translation mode 215
  - ungetting characters 664
  - updating 199, 215
  - writing characters with `fputc` 209
  - writing characters with `putc` and `putchar` 400
  - writing data items 239
  - writing lines with `puts()` 402
  - writing strings 211
- string
  - writing with `fputs` 211
- structures, TPFCS 859
- subsystem change 25
- suspend a global transaction 655
- suspend ECB if running too long 553
- suspend if resources are low 308
- suspend the calling process 507
- swapping a storage block with a data event control block (DECB) 582
- symbolic file address information 511, 586
- synchronize
  - tape 645
  - TPF global field or record 284
- syslog daemon
  - `closelog` function 48
  - `openlog` function 386
  - `syslog` function 523
- system
  - calls, general discussion 527
- system error with message 389
- system generation options test 526
- system heap storage release 443
- system interprocessor communication 504
  - `sipcc` function 504
- system ordinal number conversion 75
- system resources 306, 308
- system utilization 306, 308

## T

- table, cursor APIs 1113
- table, non-cursor APIs 881
- tape access 532
- tape control 535
- tape management functions
  - `tape_close` 534
  - `tape_cntl` 535
  - `tape_open` 537
  - `tape_read` 538
  - `tape_write` 540
  - `tasnc` 541
  - `tbspc` 542
  - `tcisc` 544
  - `tdspc_q` 546
  - `tdspc` 545
  - `topnc` 556
  - `tourc` 557
  - `toutc` 558
  - `tprdc` 641
  - `trewc` 643
  - `trsvc` 644
  - `tsync` 645
  - `twrtc` 646
- tape queue length display 546
- tape status display 545
- task, dispatching 1339
- TCP/IP network services database
  - `tpf_tcpip_message_cnt` 635
  - update message counters 635
- temporary
  - bag, creating 941, 982
  - BLOB, creating 947
  - collection, creating and assigning it a persistent identifier (PID) 935
  - copy, creating and returning its PID 929
  - key bag, creating 956
  - key set, creating 968
  - key sorted bag, creating 974, 976
  - key sorted set, creating 980
  - keyed log, creating 962
  - log, creating 986
  - sequence collection, creating 897, 1002
  - set, creating 1008
  - sorted bag collection, creating 901, 1015
  - sorted set collection, creating 1021
- temporary file
  - names 552
- temporary files 551
- terminal I/O functions
  - `assert` 15
  - `cratc` 51
  - `gets` 273
  - `puts` 402
  - `routc` 441
  - `scanf` 447
  - `tpf_help` 603
  - `wgtac_ext` 692
  - `wgtac` 691
  - `wtopc_insert_header` 704
  - `wtopc_routing_list` 706



## terminal I/O functions *(continued)*

- wtopc\_text 707
- wtopc 700
- test
  - core level for occupied condition 300
  - system generation options 526
- test for queued information, TPFxd\_sync 1382
- testing
  - data definition 1314
  - for an already defined property 1068
  - for more elements 1157
  - to see if cursor is at the end of collection 1125
  - to see if cursor points at last element 1127
  - to see if DD name is defined 1314
  - to see if PID is for a collection 1066
  - to see if the collection is empty 1147
- text
  - files 199
- text dump, setting on or off 1335
- thread application interface selection 448
- time slice an ECB 553
- time stamp creation 1272
- time-initiated entry 64, 66
- TMP\_MAX macro 552
- TPF Advanced Program-to-Program Communications (TPF/APPC) 1, 723, 793
- TPF API functions 1
  - \_\_CREDC 57, 59
  - \_\_CREMC 62
  - \_\_CRETc 64
  - \_\_CREXC 69
  - \_\_ENTDC 104
  - abort 7
  - addlc 11
  - alarm 13
  - assert 15
  - attac\_ext 21
  - attac\_id 23
  - attac 19
  - cebic\_goto\_bss 25
  - cebic\_goto\_dbi 26
  - cebic\_goto\_ssu 27
  - cebic\_restore 28
  - cebic\_save 29
  - cifrc 37
  - cinfrc\_fast\_ss 41
  - cinfrc\_fast 40
  - cinfrc 38
  - corhc 49
  - coruc 50
  - cratc 51
  - credc 57
  - cremc 62
  - cretc\_level 66
  - cretc 64
  - crexc 69
  - crosc\_entrc 71
  - crusa 73
  - csonc 75
  - dbzac 78
  - dbzdc.h 79

## TPF API functions *(continued)*

- defrc 80
- deqc 84
- detac\_ext 86
- detac\_id 88
- detac 85
- dlayc 90
- ecbptr 101
- enqc 102
- entdc 104
- evinc 108
- evnqc 109
- evntc 111
- evnwc 113
- exit 115
- face\_facs 120
- FACE 117
- FACS 117
- file\_record\_ext 161
- file\_record 158
- filec\_ext 154
- filec 152
- filnc\_ext 167
- filnc 165
- filuc\_ext 172
- filuc 170
- find\_record\_ext 181
- find\_record 178
- findc\_ext 176
- findc 174
- finhc\_ext 187
- finhc 185
- finwc\_ext 191
- finwc 189
- fiwhc\_ext 195
- fiwhc 193
- flipc 197
- gdsnc 241
- gdsrc 244
- getcc 248
- getfc 258
- getpc 265
- gets 273
- glob\_keypoint 279
- glob\_lock 280
- glob\_sync 284
- glob\_unlock 286
- glob\_update 288
- glob 277
- global 290
- gsysc 293
- inqrc 295
- keyrc\_okey 297
- keyrc 296
- levtest 300
- lockc 305
- longc 312
- mail 320
- maskc 325
- numbc 379
- pausc 387

## TPF API functions *(continued)*

perror 389  
 postc 395  
 progC 398  
 puts 402  
 raisa 404  
 rcunc 408  
 rehka 419  
 relcc 421  
 relfc 423  
 relpc 425  
 ridcc 435  
 rlcha 437  
 routc 441  
 rsysc 443  
 rvtcc 445  
 scanf 447  
 selec 448  
 select 450  
 serrc\_op\_ext 456  
 serrc\_op\_slc 458  
 serrc\_op 454  
 sipcc 504  
 snapc 508  
 sonic 511  
 strerror 517  
 swisc\_create 518  
 systc 526  
 system 527  
 tape\_access 532  
 tape\_close 534  
 tape\_cntl 535  
 tape\_open 537  
 tape\_read 538  
 tape\_write 540  
 tasnc 541  
 tbspc 542  
 tclsc 544  
 tdspc\_q 546  
 tdspc\_v 547  
 tdspc 545  
 tdtac 549  
 tmslc 553  
 topnc 556  
 tourc 557  
 toutc 558  
 tpcnc 560  
 tpf\_cresc 572  
 tpf\_decb\_create 576  
 tpf\_decb\_locate 578  
 tpf\_decb\_release 580  
 tpf\_decb\_swapblk 582  
 tpf\_decb\_validate 584  
 tpf\_dlckc 585  
 tpf\_esfac 586  
 tpf\_fa4x4c 592  
 tpf\_fac8c 587  
 tpf\_fork 594  
 tpf\_gsvac 602  
 tpf\_help 603  
 tpf\_is\_RPCServer\_auto\_restarted 607

## TPF API functions *(continued)*

tpf\_itrpc 608  
 tpf\_lemic 612  
 tpf\_movec\_EVM 618  
 tpf\_movec 616  
 tpf\_rcrfc 624  
 tpf\_RPC\_options 626  
 tpf\_snmp\_BER\_encode 632  
 tpf\_STCK 634  
 tpf\_tcpip\_message\_cnt 635  
 tpf\_tm\_getToken 636  
 tprdc 641  
 trewc 643  
 trsvc 644  
 tsync 645  
 twrtc 646  
 uatbc 657  
 unfrc\_ext 662  
 unfrc 661  
 unhka 666  
 unlkc 670  
 waitc 686  
 wgtac\_ext 692  
 wgtac 691  
 WIFEXITED 694  
 wtopc\_insert\_header 704  
 wtopc\_routing\_list 706  
 wtopc\_text 707  
 wtopc 700  
 entrc 106  
 TPF\_CALL\_BY\_NAME 562  
 tpf\_vipac 637  
 tpf\_yieldc 639

## TPF collection support

bag, creating 939, 943  
 BLOB, creating 945  
 browse support, APIs 1223  
 copy, creating and returning its PID 931  
 error handling 860  
 key bag, creating 954, 958  
 key set, creating 966, 970  
 key sorted bag, creating 972  
 key sorted set, creating 978  
 keyed log, creating 960, 964  
 log, creating 984, 988  
 restarting at cycle up 1334  
 sequence collection, creating 1000, 1004  
 set, creating 1006, 1010  
 sorted bag, creating 1013  
 sorted set, creating 1019  
 structures 859

## TPF collection support functions

T02\_add 883  
 T02\_addAllFrom 886  
 T02\_addAtCursor 1115  
 T02\_addAtIndex 888  
 T02\_addKeyPath 890  
 T02\_addRecoupIndexEntry 893  
 T02\_allElementsDo 1117  
 T02\_asSequenceCollection 897  
 T02\_associateRecoupIndexWithPID 899



TPF collection support functions *(continued)*

T02\_asSortedCollection 901  
T02\_at 903  
T02\_atBrowseKey 1224  
T02\_atBrowseKeyPut 1226  
T02\_atBrowseNewKeyPut 1228  
T02\_atCursor 1119  
T02\_atCursorPut 1121  
T02\_atCursorWithBuffer 1123  
T02\_atDSdictKey 1188  
T02\_atDSdictKeyPut 1190  
T02\_atDSdictNewKeyPut 1192  
T02\_atDSsystemKey 1194  
T02\_atDSsystemKeyPut 1196  
T02\_atDSsystemNewKeyPut 1198  
T02\_atEnd 1125  
T02\_atKey 905  
T02\_atKeyPut 907  
T02\_atKeyWithBuffer 909  
T02\_atLast 1127  
T02\_atNewKeyPut 911  
T02\_atPut 913  
T02\_atRBA 915  
T02\_atRBAPut 917  
T02\_atRBAWithBuffer 920  
T02\_atTPFKey 1200  
T02\_atTPFKeyPut 1202  
T02\_atTPFNewKeyPut 1204  
T02\_atTPFsystemKey 1206  
T02\_atTPFsystemKeyPut 1208  
T02\_atTPFsystemNewKeyPut 1210  
T02\_atWithBuffer 922  
T02\_capture 924  
T02\_changeDD 1230  
T02\_changeDS 1232  
T02\_class 1346  
T02\_convertBinPIDtoEBCDIC 1348  
T02\_convertClassName 1234  
T02\_convertEBCDICtoBinPID 1350  
T02\_convertMethodName 1236  
T02\_copyCollection 927  
T02\_copyCollectionTemp 929  
T02\_copyCollectionWithOptions 931  
T02\_createArray 933  
T02\_createArrayTemp 935  
T02\_createArrayWithOptions 937  
T02\_createBag 939  
T02\_createBagTemp 941  
T02\_createBagWithOptions 943  
T02\_createBLOB 945  
T02\_createBLOBTemp 947  
T02\_createBLOBWithOptions 949  
T02\_createCursor 1129  
T02\_createDD 1238  
T02\_createDS 1240  
T02\_createDSwithOptions 1242  
T02\_createEnv 874  
T02\_createKeyBag 954  
T02\_createKeyBagTemp 956  
T02\_createKeyBagWithOptions 958  
T02\_createKeyedLog 960

TPF collection support functions *(continued)*

T02\_createKeyedLogTemp 962  
T02\_createKeyedLogWithOptions 964  
T02\_createKeySet 966  
T02\_createKeySetTemp 968  
T02\_createKeySetWithOptions 970  
T02\_createKeySortedBag 972  
T02\_createKeySortedBagTemp 974  
T02\_createKeySortedBagWithOptions 976  
T02\_createKeySortedSet 978  
T02\_createKeySortedSetTemp 980  
T02\_createKeySortedSetWithOptions 982  
T02\_createLog 984  
T02\_createLogTemp 986  
T02\_createLogWithOptions 988  
T02\_createOptionList 990  
T02\_createPIDinventoryKey 1244  
T02\_createReadWriteCursor 1131  
T02\_createRecoupIndex 998  
T02\_createSequence 1000  
T02\_createSequenceTemp 1002  
T02\_createSequenceWithOptions 1004  
T02\_createSet 1006  
T02\_createSetTemp 1008  
T02\_createSetWithOptions 1010  
T02\_createSortedBag 1013  
T02\_createSortedBagTemp 1015  
T02\_createSortedBagWithOptions 1017  
T02\_createSortedSet 1019  
T02\_createSortedSetTemp 1021  
T02\_createSortedSetWithOptions 1023  
T02\_cursorMinus 1133  
T02\_cursorPlus 1135  
T02\_defineBrowseNameForPID 1246  
T02\_definePropertyForPID 1028  
T02\_definePropertyWithModeForPID 1031  
T02\_deleteAllPropertiesFromPID 1034  
T02\_deleteBrowseName 1248  
T02\_deleteCollection 1036  
T02\_deleteCursor 1137  
T02\_deleteDD 1250  
T02\_deleteDS 1252  
T02\_deleteEnv 876  
T02\_deletePID 1352  
T02\_deletePropertyFromPID 1038  
T02\_deleteRecoupIndex 1040  
T02\_deleteRecoupIndexEntry 1041  
T02\_first 1139  
T02\_getAllPropertyNamesFromPID 1043  
T02\_getBLOB 1045  
T02\_getBLOBWithBuffer 1047  
T02\_getBrowseDictPID 1254  
T02\_getClassAttributes 1256  
T02\_getClassDocumentation 1258  
T02\_getClassInfo 1260  
T02\_getClassNames 1262  
T02\_getClassTree 1264  
T02\_getCollectionAccessMode 1049  
T02\_getCollectionAttributes 1266  
T02\_getCollectionKeys 1051  
T02\_getCollectionName 1268

## TPF collection support functions *(continued)*

T02\_getCollectionParts 1270  
 T02\_getCollectionType 1053  
 T02\_getCreateTime 1272  
 T02\_getCurrentKey 1141  
 T02\_getCurrentKeyWithBuffer 1143  
 T02\_getDDAttributes 1274  
 T02\_getDirectoryForRRN 1276  
 T02\_getDRprotect 1055  
 T02\_getDSAttributes 1278  
 T02\_getDSdictPID 1212  
 T02\_getDSnameForPID 1280  
 T02\_getErrorCode 877  
 T02\_getErrorText 879  
 T02\_getKeyPathAttributes 1281  
 T02\_getListDDnames 1283  
 T02\_getListDSCollections 1285  
 T02\_getListDSnames 1287  
 T02\_getListUsers 1289  
 T02\_getMaxDataLength 1057  
 T02\_getMaxKeyLength 1058  
 T02\_getMethodDocumentation 1291  
 T02\_getMethodNames 1293  
 T02\_getNumberOfRecords 1295  
 T02\_getPathInfoFor 1296  
 T02\_getPIDforBrowseName 1299  
 T02\_getPIDinventoryEntry 1301  
 T02\_getPIDinventoryPID 1303  
 T02\_getPropertyValueFromPID 1060  
 T02\_getRecordAttributes 1305  
 T02\_getRecoupIndex 1307  
 T02\_getRecoupIndexForPID 1310  
 T02\_getSortFieldValues 1062  
 T02\_getTPFDictPID 1214  
 T02\_getUserAttributes 1312  
 T02\_includes 1064  
 T02\_index 1145  
 T02\_isCollection 1066  
 T02\_isDDdefined 1314  
 T02\_isEmpty 1147  
 T02\_isExtended 1316  
 T02\_isPropertyDefinedForPID 1068  
 T02\_isTemp 1070  
 T02\_key 1149  
 T02\_keyWithBuffer 1151  
 T02\_last 1153  
 T02\_locate 1155  
 T02\_makeEmpty 1072  
 T02\_maxEntry 1074  
 T02\_migrateCollection 1318  
 T02\_migrateDS 1320  
 T02\_more 1157  
 T02\_next 1159  
 T02\_nextPut 1161  
 T02\_nextRBAfor 1163  
 T02\_nextWithBuffer 1165  
 T02\_peek 1168  
 T02\_peekWithBuffer 1170  
 T02\_previous 1172  
 T02\_previousWithBuffer 1174  
 T02\_readOnly 1076

## TPF collection support functions *(continued)*

T02\_reclaimPID 1322  
 T02\_reconstructCollection 1323  
 T02\_recoupCollection 1325  
 T02\_recoupDS 1327  
 T02\_recoupPT 1329  
 T02\_recreateDS 1330  
 T02\_remove 1176  
 T02\_removeBrowseKey 1332  
 T02\_removeDSdictKey 1215  
 T02\_removeDSsystemKey 1217  
 T02\_removeIndex 1078  
 T02\_removeKey 1080  
 T02\_removeKeyPath 1082  
 T02\_removeRBA 1084  
 T02\_removeRecoupIndexFromPID 1086  
 T02\_removeTPFKey 1219  
 T02\_removeTPFsystemKey 1221  
 T02\_removeValue 1088  
 T02\_removeValueAll 1090  
 T02\_replaceBLOB 1092  
 T02\_reset 1178  
 T02\_restart 1334  
 T02\_restore 1094  
 T02\_restoreAsTemp 1096  
 T02\_restoreWithOptions 1098  
 T02\_setClass 1354  
 T02\_setCollectionAccessMode 1101  
 T02\_setDRprotect 1103  
 T02\_setGetTextDump 1335  
 T02\_setKeyPath 1180  
 T02\_setMethodTrace 1337  
 T02\_setPositionIndex 1182  
 T02\_setPositionValue 1184  
 T02\_setReadOnly 1105  
 T02\_setSize 1107  
 T02\_size 1109  
 T02\_taskDispatch 1339  
 T02\_validateCollection 1340  
 T02\_validateKeyPath 1342  
 T02\_writeNewBLOB 1111

## TPF file system support

C language functions  
   path name rules 1408  
 programming interfaces  
   list of functions 1405  
   pointers and file descriptors 1409  
   security 1409

## TPF Internet mail functions

mail 320

## TPF MQSeries functions

MQBACK 333  
 MQCLOSE 335  
 MQCMIT 337  
 MQCONN 339  
 MQDISC 341  
 MQGET 343  
 MQINQ 349  
 MQOPEN 355  
 MQPUT 359  
 MQPUT1 365

## TPF MQSeries functions *(continued)*

- MQSET 372
- TPF\_BAL\_FN type 106, 562
- TPF\_BAL\_FN\_PTR type 106, 562
- TPF\_CALL\_BY\_NAME macro 562
- TPF\_CALL\_BY\_NAME\_STUB macro 562
- TPF\_FSDD\_APPEND type 1416
- TPF\_FSDD\_CLOSE type 1419
- TPF\_FSDD\_GET type 1420
- TPF\_FSDD\_OPEN type 1422
- TPF\_FSDD\_POLL type 1425
- TPF\_FSDD\_POLL\_CLEAN type 1427
- TPF\_FSDD\_PUT type 1429
- TPF\_FSDD\_RESIZE type 1432
- TPF\_FSDD\_SIZE type 1434
- TPF\_FSDD\_SYNC type 1436
- tpf\_vipac function 637
- tpf\_yieldc function 639
- TPF-supplied user names 1410
- TPF/APPC basic conversation functions
  - tppc\_activate\_on\_confirmation 737
  - tppc\_activate\_on\_receipt 741
  - tppc\_allocate 745
  - tppc\_confirm 750
  - tppc\_confirmed 753
  - tppc\_deallocate 755
  - tppc\_flush 759
  - tppc\_get\_attributes 761
  - tppc\_get\_type 763
  - tppc\_post\_on\_receipt 765
  - tppc\_prepare\_to\_receive 768
  - tppc\_receive 771
  - tppc\_request\_to\_send 777
  - tppc\_send\_data 779
  - tppc\_send\_error 782
  - tppc\_test 786
  - tppc\_wait 789
  - general programming considerations 735
  - general syntax 723
  - valid return codes 731
  - valid verbs and keywords 724

## TPF/APPC mapped conversation functions

- cmaccp 798
- cmallc 800
- cmcfm 803
- cmcfmd 805
- cmdeal 807
- cmecs 810
- cmemn 812
- cmesl 816
- cmflus 818
- cminit 820
- cmptr 823
- cmrcv 825
- cmrts 829
- cmsdt 831
- cmsed 833
- cmsend 835
- cmserr 838
- cmsmn 842
- cmspln 844

## TPF/APPC mapped conversation functions *(continued)*

- cmsptr 846
- cmsrc 848
- cmssl 850
- cmsst 852
- cmstpn 854
- cmtrts 856
- characteristics 794
- conversation states 794
- mapped conversation interface overview 793
- side information table 795
- valid return codes 796
- TPFAR database support structure
  - attach 78
  - detach 79
- transaction anchor table control 530
- transaction branch
  - xa\_rollback 718
  - commit work 708
  - end work 710
  - get a list of prepared transaction branches 716
  - prepare to commit 714
  - roll back work 718
  - start 720
- traps for SNMP applications, sending 608
- truncate a file 237
- type definitions 861, 874
- type value, getting the collections 1053

## U

- unhold a file record 661
- unhold a file record with extended options 662
- unhold a resource 50
- unhold file record 408
- unhook core block 666
- unlock
  - a resource 670
  - TPF global field or record 286
- update an existing cache entry 673
- update message counters for TCP/IP applications 635
- update TPF global field or record 288
- user ID 35
  - chown 35
- user names, TPF-supplied 1410

## V

- validating a collection 1340
- validating a data event control block (DECB) 584
- value, retrieving for a property 1060
- verify condition 15
  - print diagnostic message 15

## W

- wait for a signal 388
- wait for event completion with signal awareness 627
- wait for outstanding I/O completion 686
- wait for status information from a child process 684

- working directory
  - chdir 30
  - path name 252
- working storage block release 421
- wrapping of output 402
- write a record to general tape 540, 646
- write buffer to file 144
- write core block images to the external device 1386
- write data from the malloc area to an external device,  
TPFxd\_write 1384
- write real-time tape record 558
- write real-time tape record and release storage  
block 557
- writing records to an external device 1357

## Y

- yield control 639

## Z

- ZBROW command
  - delete a browse name 1248
  - get a PID for a browse name 1299





File Number: S370/30XX-40  
Program Number: 5748-T14



Printed in the United States of America  
on recycled paper containing 10%  
recovered post-consumer fiber.

SH31-0121-09

