

Transaction Processing Facility



# Transmission Control Protocol/Internet Protocol

*Version 4 Release 1*



Transaction Processing Facility



# Transmission Control Protocol/Internet Protocol

*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 xvii.

**Thirteenth Edition (June 2002)**

This is a major revision of, and obsoletes, SH31-0120-11 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 1996, 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> . . . . .	xiii
<b>Tables</b> . . . . .	xv
<b>Notices</b> . . . . .	xvii
Trademarks . . . . .	xvii
<b>About This Book</b> . . . . .	xix
Before You Begin . . . . .	xix
Who Should Read This Book . . . . .	xix
How This Book Is Organized. . . . .	xix
Conventions Used in the TPF Library. . . . .	xx
Related Information . . . . .	xxi
IBM Transaction Processing Facility (TPF) 4.1 Books. . . . .	xxi
IBM Transmission Control Protocol/Internet Protocol (TCP/IP) Books . . . . .	xxi
IBM Common Link Access to Workstation (CLAW) Books . . . . .	xxi
IBM 3172 Interconnect Controller Books . . . . .	xxi
Online Information . . . . .	xxii
How to Send Your Comments . . . . .	xxii

---

## Part 1. TCP/IP Support Overview. . . . . 1

<b>Networking Protocols Introduction</b> . . . . .	3
TPF System Support of Transmission Control Protocol/Internet Protocol . . . . .	3
TCP/IP Native Stack Support . . . . .	4
<b>TCP/IP Network Overview</b> . . . . .	7
Network Requirements . . . . .	7
Industry-Standard Transport Application Programming Interface (API) . . . . .	7
Open Network Connectivity . . . . .	7
Client/Server Environment . . . . .	7
Enhanced Role in the Internet. . . . .	7
Porting Socket Applications. . . . .	7

---

## Part 2. TCP/IP Offload Support. . . . . 9

<b>TCP/IP Internals</b> . . . . .	11
IBM 3172 Model 3 Interconnect Controller Overview . . . . .	11
Configuration Characteristics of TCP/IP Offload Devices. . . . .	12
Data Flow between the Offload Device and the TPF System . . . . .	12
Sample Transmission Control Protocol/Internet Protocol Network . . . . .	13
Components of TCP/IP Support. . . . .	16
Outbound Message Flow through the Socket/CLAW Interfaces . . . . .	17
Inbound Message Flow through the Socket/CLAW Interfaces . . . . .	18
Nonsocket User Exits . . . . .	20
Nonsocket Activation. . . . .	20
Nonsocket Connect . . . . .	20
Nonsocket Deactivation. . . . .	20
Nonsocket Message . . . . .	21
<b>TPF Control Block Structures</b> . . . . .	23
CLAW Device Table (CDT) and Related Control Block Structures . . . . .	23

Contents of the CLAW Device Table . . . . .	23
Related Control Block Structures . . . . .	23
Determining the Value for the CLAWADP Parameter . . . . .	23
File Descriptor Table and Related Control Block Structures. . . . .	24
Contents of the File Descriptor Table . . . . .	24
Related Control Block Structures . . . . .	24
Determining the Value for the CLAWFD Parameter . . . . .	25
Internet Protocol Address Table . . . . .	25
Contents of the Internet Protocol Address Table . . . . .	26
Related Control Block Structures . . . . .	26
Determining the Value for the CLAWIP Parameter . . . . .	26
Maximum Value for the IP Parameter. . . . .	26
Storage Considerations . . . . .	26
SNAKEY Parameters . . . . .	27
Miscellaneous Control Block Structures . . . . .	27
Socket Thread Control Blocks . . . . .	27
Calculating the Approximate Total Number of TCP/IP Bytes . . . . .	27
Example Calculating the Approximate Total Number of TCP/IP Bytes . . . . .	27
Tuning and Performance . . . . .	28
<b>Operator Procedures for TCP/IP Offload Support . . . . .</b>	<b>29</b>
Configuring a TCP/IP System . . . . .	29
Defining the CLAW Host Name . . . . .	29
Considerations for IBM 3172 Model 3 Interconnect Controllers . . . . .	29
Defining CLAW Workstations for a TPF Host Processor . . . . .	29
Considerations for IBM 3172 Model 3 Interconnect Controllers . . . . .	30
Activating and Deactivating CLAW Workstations. . . . .	30
Displaying Information about TCP/IP Support. . . . .	31
Deleting a CLAW Workstation . . . . .	31
Moving a CLAW Workstation from One TPF Host Processor to Another . . . . .	31
Performing a Hardware Switchover . . . . .	32
Using the CLAW Data Trace Function . . . . .	32
Starting the CLAW Data Trace Function. . . . .	32
Stopping the CLAW Data Trace Function . . . . .	32
Using the CLAW Process Trace Function . . . . .	33
Starting the CLAW Process Trace Function . . . . .	33
Stopping the CLAW Process Trace Function . . . . .	33
Resetting the ZCLAW Command Lock . . . . .	33

---

## **Part 3. TCP/IP Native Stack Support . . . . . 35**

<b>TCP/IP Native Stack Support Internals . . . . .</b>	<b>39</b>
TCP/IP Layers . . . . .	39
Using CDLC IP Routers . . . . .	41
Configuration Characteristics of CDLC IP Routers . . . . .	41
Data Flow between the CDLC IP Router and the TPF System . . . . .	41
Sample TCP/IP Networks . . . . .	41
Using OSA-Express Support . . . . .	43
Configuration Characteristics of the OSA-Express Card . . . . .	43
Data Flow between the OSA-Express Card and the TPF System . . . . .	44
Components of TCP/IP Native Stack Support. . . . .	44
Policy Agent . . . . .	46
Outbound Message Flow . . . . .	46
Outbound Message Flow for CDLC . . . . .	47
Outbound Message Flow for OSA-Express . . . . .	48
Inbound Message Flow . . . . .	48

<b>TPF Control Block Structures</b> . . . . .	51
Socket Block Table Structure . . . . .	51
Defining the Socket Block Table . . . . .	51
CDLC IP Configuration Record . . . . .	51
Defining the CDLC IP Network Configuration . . . . .	52
CDLC IP CCW Area Table . . . . .	52
Defining CDLC IP CCW Area Table Resources . . . . .	52
OSA Configuration Record . . . . .	52
OSA Control Block Table . . . . .	52
OSA Shared IP Address Table . . . . .	52
OSA Read Buffers . . . . .	53
Defining OSA Read Buffers . . . . .	53
IP Message Table . . . . .	53
Defining the IP Message Table . . . . .	54
IP Routing Table . . . . .	54
Defining the IP Routing Table . . . . .	54
Tuning TCP/IP Native Stack Support . . . . .	54
Tuning Major Control Block Structures . . . . .	55
Tuning the IP over CDLC Link Layer . . . . .	55
Tuning the IP Network . . . . .	55
Performance . . . . .	56
TCP/IP Network Configurations . . . . .	56
Selecting the Local IP Address . . . . .	56
Defining Gateways . . . . .	58
<b>Operator Procedures for TCP/IP Native Stack Support</b> . . . . .	59
Configuring a TPF System . . . . .	59
Enabling TCP/IP Native Stack Support . . . . .	60
Local IP Addresses . . . . .	60
Default Local IP Address . . . . .	61
Maximum Packet Size . . . . .	61
Types of TPF Local IP Addresses . . . . .	62
CDLC Addresses . . . . .	62
Real OSA IP Addresses . . . . .	63
Virtual IP Addresses (VIPAs) . . . . .	63
CDLC IP Connections . . . . .	65
Defining CDLC IP Connections . . . . .	65
Activating and Deactivating CDLC IP Routers . . . . .	65
Deleting CDLC IP Routers . . . . .	66
OSA-Express Connections . . . . .	66
Defining OSA-Express Cards to the Processor . . . . .	66
Defining OSA-Express Connections to TPF . . . . .	67
Activating and Deactivating OSA-Express Connections . . . . .	67
Displaying OSA-Express Connections . . . . .	67
Deleting OSA-Express Connections . . . . .	67
Gateways . . . . .	68
Routing Information Protocol . . . . .	68
How Network and Processor Failures Affect VIPAs . . . . .	68
Swinging VIPAs to an Alternate OSA-Express Connection . . . . .	68
Moving VIPAs from One Processor to Another . . . . .	68
Workload Balancing Using Movable VIPAs . . . . .	69
Configuration Examples . . . . .	71
Managing IP Routing Table Entries . . . . .	74
Displaying TCP/IP Native Stack Support . . . . .	75
Starting and Stopping the IP Trace Function . . . . .	75
Displaying IP Trace Information . . . . .	75

Using the Individual IP Trace Function . . . . .	76
Displaying Individual IP Trace Tables . . . . .	77
Deactivating Sockets . . . . .	77
Displaying Socket Control Block Information . . . . .	78
<b>Socket Application Design Considerations . . . . .</b>	<b>79</b>
Sharing Sockets . . . . .	79
Using Existing Socket Applications . . . . .	79
New Socket Options Supported . . . . .	80
Send Buffer and Receive Buffer Sizes . . . . .	80
Timeouts . . . . .	81
Low-Water Marks . . . . .	81
activate_on_accept API . . . . .	82
Local Sockets . . . . .	82
<b>Simple Network Management Protocol Agent Support . . . . .</b>	<b>85</b>
SNMP Overview . . . . .	85
SNMP Manager . . . . .	85
SNMP Agent . . . . .	85
Management Information Base (MIB) . . . . .	85
Interaction between SNMP Components . . . . .	86
Protocol Data Units (PDUs) . . . . .	87
Structure and Fields of SNMP PDUs . . . . .	87
TPF SNMP Agent Support . . . . .	89
Implementing Management Information Base-II (MIB-II) . . . . .	90
Processing SNMP Requests . . . . .	91
Message Processing . . . . .	91
User MIB Variables . . . . .	93
SNMP Traps . . . . .	93
Installing TPF SNMP Agent Support . . . . .	96
Installing and Defining TPF TCP/IP Native Stack Support . . . . .	96
Installing the SNMP Agent . . . . .	96
Creating the SNMP Configuration File . . . . .	96
Coding the UCOM and UMIB User Exits . . . . .	98
Defining and Starting the SNMP Agent Server . . . . .	98
Defining IP Routing Table Entries . . . . .	99
Defining the TPF System to the SNMP Manager . . . . .	99
<b>Domain Name System Support . . . . .</b>	<b>101</b>
DNS Server . . . . .	101
TPF Host Name Table . . . . .	101
IP Address Selection . . . . .	102
DNS Client . . . . .	103
<b>Internet Security . . . . .</b>	<b>105</b>
Denial-of-Service Attacks . . . . .	105
Packet Filtering . . . . .	105
Packet Filtering Rules File Syntax . . . . .	106
Considerations for Packet Filtering Rules . . . . .	108
Examples of Packet Filtering Rules . . . . .	108
Problem Diagnosis . . . . .	109
<b>TCP/IP Network Services Database Support . . . . .</b>	<b>111</b>
Quality of Service . . . . .	111
Data Collection and Reduction . . . . .	112
Message Counts by Application . . . . .	112

TCP/IP Network Services Database File . . . . .	113
TCP/IP Network Services Database File Syntax . . . . .	113
TCP/IP Network Services Database File Example. . . . .	114

---

**Part 4. Socket Application Programming Interface Overview. . . . . 117**

<b>Socket Overview . . . . .</b>	<b>119</b>
Sockets . . . . .	119
Types of Sockets Supported by TCP/IP . . . . .	119
Socket Address for the Internet Domain . . . . .	119
Port Numbers . . . . .	120
Standard Dotted Decimal Formats . . . . .	120
Mapping Address Parts . . . . .	120
Integer Byte Order Conversion . . . . .	120
Blocking and Nonblocking . . . . .	121
Out-Of-Band Data . . . . .	121
TPF Socket Application Programming Interface (API) Support . . . . .	122
Socket API Functions Using TCP/IP Offload Support . . . . .	122
Socket API Functions Using TCP/IP Native Stack Support . . . . .	122
Socket User Exits . . . . .	122
Socket Accept for TCP/IP Offload Support . . . . .	123
Socket Activation. . . . .	123
Socket Connect . . . . .	123
Socket Cycle-Up When Using TCP/IP Offload Support . . . . .	123
Socket Deactivation. . . . .	123
Socket System Error . . . . .	124
TCP/IP Native Stack Support Accept Connection . . . . .	124
Select TCP/IP Support . . . . .	124
Socket Cycle-Up When Using TCP/IP Native Stack Support . . . . .	124
Full-Duplex Socket Support . . . . .	124
Using activate_on_receipt . . . . .	125
Socket Sweeper Support to Close Inactive Sockets . . . . .	125
<b>Sample Socket Sessions . . . . .</b>	<b>127</b>
Function Calls Used in a Sample TCP Session . . . . .	127
Using the activate_on_receipt Function Call . . . . .	129
Function Calls Used in a Sample UDP Session . . . . .	131
Main Socket Function Calls . . . . .	132
<b>Socket Application Programming Interface Functions Reference . . . . .</b>	<b>137</b>
General Function Information . . . . .	137
accept — Accept a Connection Request . . . . .	138
activate_on_accept — Activate a Program When the Client Connects . . . . .	141
activate_on_receipt — Activate a Program after Data Received . . . . .	144
activate_on_receipt_with_length — Activate a Program after Data of Specified Length Received . . . . .	148
bind — Bind a Local Name to the Socket. . . . .	152
close — Shut Down a Socket . . . . .	155
connect — Request a Connection to a Remote Host . . . . .	157
gethostbyaddr — Get Host Information for IP Address . . . . .	160
gethostbyname — Get IP Address Information by Host Name . . . . .	162
gethostid — Return Identifier of Current Host . . . . .	164
gethostname — Return Host Name . . . . .	166
getpeername — Return the Name of the Peer . . . . .	168
getservbyname — Get Server Port by Name . . . . .	170
getservbyport — Get Server Name by Port . . . . .	172

getsockname — Return the Name of the Local Socket . . . . .	174
getsockopt — Return Socket Options . . . . .	176
htonl — Translate a Long Integer. . . . .	180
htons — Translate a Short Integer . . . . .	181
inet_addr — Construct Internet Address from Character String . . . . .	182
inet_ntoa — Return Pointer to a String in Dotted Decimal Notation . . . . .	184
ioctl — Perform Special Operations on Socket . . . . .	185
listen — Complete Binding, Create Connection Request Queue . . . . .	189
ntohl — Translate a Long Integer. . . . .	191
ntohs — Translate a Short Integer . . . . .	192
read — Read Data on a Socket . . . . .	193
recv — Receive Data on a Connected Socket . . . . .	196
recvfrom — Receive Data on Connected/Unconnected Socket . . . . .	199
recvmsg — Receive Message on Connected/Unconnected Socket . . . . .	202
select — Monitor Read, Write, and Exception Status . . . . .	205
send — Send Data on a Connected Socket . . . . .	206
sendmsg — Send Message on a Socket . . . . .	210
sendto — Send Data on an Unconnected Socket. . . . .	212
setsockopt — Set Options Associated with a Socket. . . . .	216
shutdown — Shut Down All or Part of a Duplex Connection . . . . .	220
sock_errno — Return the Error Code Set by a Socket Call . . . . .	222
socket — Create an Endpoint for Communication. . . . .	223
write — Write Data on a Connected Socket . . . . .	226
writv — Write Data on a Connected Socket . . . . .	229

---

**Part 5. Operator Procedures for Internet Server Applications . . . . . 233**

<b>Operator Procedures for the Internet Daemon . . . . .</b>	<b>235</b>
Internet Daemon . . . . .	235
Internet Daemon Configuration File . . . . .	235
Internet Daemon Control . . . . .	236
Internet Server Application . . . . .	237
Internet Server Application Control . . . . .	238
<b>Trivial File Transfer Protocol (TFTP) Server . . . . .</b>	<b>239</b>
Adding the Trivial File Transfer Protocol (TFTP) Server. . . . .	239
Directives for the TFTP Configuration File . . . . .	240
Creating the TFTP Configuration File . . . . .	241
Transferring and Maintaining the TFTP Configuration File . . . . .	241
<b>File Transfer Protocol (FTP) Server . . . . .</b>	<b>243</b>
FTP Server LOG File . . . . .	243
Adding the File Transfer Protocol (FTP) Server . . . . .	243
<b>Syslog Daemon . . . . .</b>	<b>245</b>
Files . . . . .	245
Syslog Daemon Configuration File . . . . .	246
Modifying the Syslog Daemon Configuration File . . . . .	249
Adding the Syslog Daemon Server . . . . .	249
Operating the Syslog Daemon. . . . .	250
Starting the Syslog Daemon . . . . .	250
Stopping the Syslog Daemon . . . . .	250
Offloading Log Files . . . . .	251
Diagnosing Syslog Daemon Configuration Problems. . . . .	251
Application Considerations . . . . .	251

<b>TPF Internet Mail Server Support</b> . . . . .	253
TPF Internet Mail Server Overview . . . . .	253
Mail Database Layout . . . . .	255
Recoup Considerations for the Mail Database . . . . .	258
TPF Internet Mail Server Configuration Files . . . . .	259
SMTP Configuration File Parameters . . . . .	259
IMAP/POP Configuration File Parameters . . . . .	262
TPF Configuration File Parameters . . . . .	263
Access List Configuration Parameters . . . . .	264
TPF Internet Mail Server Administrator or Operator Tasks . . . . .	265
Configuring the TPF System for TPF Internet Mail Server Support . . . . .	265
Adding a Domain to an Existing TPF Internet Mail Server Configuration . . . . .	267
Adding New Users to an Existing TPF Internet Mail Server Configuration . . . . .	268
Controlling the TPF Internet Mail Servers . . . . .	268
Managing Client Mailboxes . . . . .	269
TPF Internet Mail Server Client Tasks . . . . .	269

---

**Part 6. Secure Sockets Layer (SSL) Support** . . . . . 271

<b>Secure Sockets Layer (SSL) Support</b> . . . . .	273
SSL_accept . . . . .	274
SSL_aor . . . . .	275
SSL_check_private_key . . . . .	277
SSL_connect . . . . .	279
SSL_CTX_check_private_key . . . . .	280
SSL_CTX_free . . . . .	281
SSL_CTX_load_and_set_client_CA_list . . . . .	282
SSL_CTX_load_verify_locations . . . . .	283
SSL_CTX_new . . . . .	285
SSL_CTX_new_shared . . . . .	287
SSL_CTX_set_cipher_list . . . . .	289
SSL_CTX_set_client_CA_list . . . . .	292
SSL_CTX_set_default_passwd_cb_userdata . . . . .	293
SSL_CTX_set_verify . . . . .	294
SSL_CTX_use_certificate_chain_file . . . . .	296
SSL_CTX_use_certificate_file . . . . .	298
SSL_CTX_use_PrivateKey_file . . . . .	300
SSL_CTX_use_RSAPrivateKey_file . . . . .	302
SSL_free . . . . .	304
SSL_get_cipher . . . . .	305
SSL_get_error . . . . .	307
SSL_get_peer_certificate . . . . .	309
SSL_get_session . . . . .	310
SSL_get_verify_result . . . . .	311
SSL_get_version . . . . .	313
SSL_library_init . . . . .	314
SSL_load_and_set_client_CA_list . . . . .	315
SSL_load_client_CA_file . . . . .	316
SSL_new . . . . .	317
SSL_pending . . . . .	318
SSL_read . . . . .	319
SSL_renegotiate . . . . .	320
SSL_set_cipher_list . . . . .	321
SSL_set_client_CA_list . . . . .	324
SSL_set_fd . . . . .	325
SSL_set_session . . . . .	326

SSL_set_verify . . . . .	327
SSL_shutdown . . . . .	329
SSL_use_certificate_file . . . . .	330
SSL_use_PrivateKey_file. . . . .	332
SSL_use_RSAPrivateKey_file . . . . .	333
SSL_write . . . . .	334
SSLv2_client_method . . . . .	335
SSLv2_server_method . . . . .	336
SSLv23_client_method . . . . .	337
SSLv23_server_method . . . . .	338
SSLv3_client_method . . . . .	339
SSLv3_server_method . . . . .	340
TLSv1_client_method . . . . .	341
TLSv1_server_method . . . . .	342
<b>Appendix A. CLAW Trace Postprocessor . . . . .</b>	<b>343</b>
Sample JCL for the CLAW Data Trace Postprocessor . . . . .	343
CLAW Data Trace Postprocessor. . . . .	344
Sample JCL for the CLAW Process Trace Postprocessor . . . . .	350
CLAW Process Trace Postprocessor . . . . .	350
<b>Appendix B. ISO-C Structures Called by Socket API Functions . . . . .</b>	<b>363</b>
Structures Defined in the socket.h Header File. . . . .	363
Additional Structures . . . . .	363
<b>Appendix C. Socket Error Return Codes . . . . .</b>	<b>365</b>
<b>Appendix D. Sample Application Driver Code . . . . .</b>	<b>367</b>
activate_on_receipt Transmission Control Protocol (TCP) Server . . . . .	367
activate_on_receipt Transmission Control Protocol (TCP) Child Server . . . . .	369
Transmission Control Protocol (TCP) Server . . . . .	371
Transmission Control Protocol (TCP) Client . . . . .	373
User Datagram Protocol (UDP) Server. . . . .	376
User Datagram Protocol (UDP) Client . . . . .	378
<b>Appendix E. TCP/IP Restricted CLAW C Functions: Reference . . . . .</b>	<b>385</b>
claw_accept — Accept a CONNECT Request from the Workstation . . . . .	386
claw_closeadapter — Terminate CLAW Activity on Subchannel Pair . . . . .	388
claw_connect — Initiate a Request to Open a Logical Link . . . . .	390
claw_disconnect — Remove a Logical Link from an Adapter. . . . .	393
claw_end — Terminate All CLAW Activity . . . . .	395
claw_initialization — Prepare for CLAW Activity . . . . .	396
claw_openadapter — Initialize an Adapter . . . . .	397
claw_query — Get the Status of CLAW Adapter or Logical Links . . . . .	400
claw_send — Send a Message on an Active Logical Link. . . . .	403
CLAW Return Codes . . . . .	405
<b>Appendix F. Using the Internet Protocol Trace Facility. . . . .</b>	<b>407</b>
About the IP Trace Table. . . . .	407
Starting the IP Trace Facility and Specifying Which Data to Trace. . . . .	407
Stopping the IP Trace Facility . . . . .	408
Defining How Much Data to Store in the IP Trace Table . . . . .	408
Writing the IP Trace Table to a Real-Time Tape . . . . .	409
Displaying Information about the IP Trace Facility. . . . .	409
Displaying the IP Trace Table Online . . . . .	409
Creating a Compacted Display of the IP Trace Table . . . . .	410

Creating a Formatted Display of the IP Trace Table . . . . .	410
Using the Offline IPTPRT Utility to Create an IPTPRT Report . . . . .	410
Sample JCL for the IPTPRT Utility . . . . .	411
Defining the IPTPRT Report . . . . .	411
IPTPRT Messages . . . . .	421
Including the IP Trace Table in System Error Dumps . . . . .	421
<b>Appendix G. Management Information Base Variables.</b> . . . . .	423
MIB Variable Types . . . . .	423
<b>Index</b> . . . . .	429



# Figures

1.	TCP/IP Sample Computer Network . . . . .	4
2.	TCP/IP Native Stack Support Sample Computer Network Using Channel Data Link Control (CDLC) . . . . .	5
3.	TCP/IP Native Stack Support Sample Computer Network Using the OSA-Express Interface. . . . .	6
4.	TCP/IP Native Stack Support Sample Computer Network Using Combined CDLC and OSA-Express Interfaces. . . . .	6
5.	Two IBM 3172 Model 3 Interconnect Controller Offload Devices Connected to a Host Transaction Processing Facility Processor . . . . .	12
6.	TPF Implementation of the IBM 3172 Model 3 Interconnect Controller with the Offload Program . . . . .	13
7.	Network Overview . . . . .	15
8.	TCP/IP Support Components Overview . . . . .	16
9.	Outbound Message Flow . . . . .	18
10.	Inbound Message Flow . . . . .	19
11.	TCP/IP Layers. . . . .	39
12.	TCP/IP Layers with OSA-Express Support . . . . .	40
13.	One TPF Host Connected to One IP Network . . . . .	42
14.	Three TPF Hosts Connected to Two IP Networks . . . . .	43
15.	TCP/IP Native Stack Support Components . . . . .	45
16.	TCP/IP Native Stack Support Sample Computer Network Using OSA-Express with Static and Movable VIPAs . . . . .	72
17.	TCP/IP Native Stack Support Sample Computer Network Using OSA-Express with Swinging VIPAs . . . . .	73
18.	TCP/IP Native Stack Support Sample Computer Network Using OSA-Express with Movable VIPAs . . . . .	74
19.	Interaction between SNMP Components . . . . .	86
20.	SNMP Agent Sending a Trap Message. . . . .	87
21.	IPTPRT Report Example . . . . .	110
22.	/etc/services File Example . . . . .	115
23.	Sample Socket Session Using TCP Protocol . . . . .	128
24.	Using the activate_on_receipt Function Call. . . . .	130
25.	Sample Socket Session Using UDP Protocol . . . . .	131
26.	Relationship of the Internet Daemon Monitor and Internet Daemon Listeners . . . . .	237
27.	Syslog Daemon Operation . . . . .	245
28.	Sample /etc/syslog.conf File . . . . .	248
29.	TPF Internet Mail Server Overview. . . . .	254
30.	TPF Internet Mail Server Database — Accessing Mail. . . . .	255
31.	TPF Internet Mail Server Database — Accepting and Delivering Mail . . . . .	257
32.	Sample JCL for the CLAW Data Trace Postprocessor . . . . .	343
33.	Sample Output from the CLAW Data Trace Postprocessor . . . . .	346
34.	Sample JCL for the CLAW Process Trace Postprocessor . . . . .	350
35.	Sample Output from the CLAW Process Trace Postprocessor . . . . .	352
36.	JCL for the IPTPRT Utility . . . . .	411
37.	Compacted IPTPRT Report . . . . .	418
38.	Formatted IPTPRT Report . . . . .	421



---

## Tables

1. Assumed TCP/IP Values . . . . .	27
2. GETREQUEST, GETNEXTREQUEST, SETREQUEST, GETRESPONSE PDU Format . . . . .	88
3. Trap PDU Format . . . . .	88
4. Variable Binding Format . . . . .	89
5. SNMP Network Management Protocol Packet . . . . .	91
6. Message Definitions for TPF TCP/IP Server Applications . . . . .	115
7. Socket Types and Associated Data . . . . .	119
8. Relationship of STATE and ACT Parameters with the TPF System State . . . . .	238
9. Socket Error Return Codes . . . . .	365
10. CLAW Query Level 1 Buffer . . . . .	400
11. CLAW Query Level 2 Buffer . . . . .	400
12. CLAW Query Level 3 Buffer . . . . .	401
13. CLAW Return Codes Defined for CLAW Functions . . . . .	405
14. MIB Variables Supported by the TPF System . . . . .	423



---

## 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:

AIX  
Enterprise Systems Connection Architecture  
ESCON  
IBM  
MQSeries  
OS/2  
OS/390  
System/390  
VisualAge.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

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

---

## About This Book

This book provides an overview of the Internet, Transaction Processing Facility (TPF) support of Transmission Control Protocol/Internet Protocol (TCP/IP), and the socket application programming interface (API) functions used by application programmers.

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*.

---

## Before You Begin

You should have a basic knowledge of the TPF system and some knowledge of system communication and network protocols.

---

## Who Should Read This Book

This book is written primarily for application programmers who will be accessing the API functions provided.

This book can also be used by system programmers and system operators who need an overview of this support and more detailed information for their particular area of expertise.

---

## How This Book Is Organized

This book has the following parts:

- Part 1 of this book provides an overview of the Internet and information concerning the TPF system implementation of TCP/IP including internal structures and procedural information for performing specific operations.
- Part 2 of this book provides TCP/IP offload support information.
- Part 3 of this book provides TCP/IP native stack support information.
- Part 4 of this book provides an overview of sockets, definitions of terms related to sockets, procedural information, and examples. Part 4 also contains a detailed reference section containing the socket functions implemented by the TPF system.
- Part 5 of this book provides information about operator procedures for Internet server applications.
- Part 6 of this book provides information about Secure Sockets Layer (SSL) support.
- The appendixes contain the following information:
  - Appendix A - sample output from the postprocessor used to print ZCLAW trace data
  - Appendix B - sample ISO-C structures used by the socket API
  - Appendix C - socket error return codes
  - Appendix D - sample drivers that provide examples of code that application programmers can customize for their own use

- Appendix E - restricted TCP/IP Common Link Access to Workstation (CLAW) ISO-C functions.
- Appendix F - Internet protocol (IP) trace facility information
- Appendix G - Management Information Base (MIB) variables.

In addition, an index is provided to help you find information in this book.

---

## 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> <pre> ...+...1...+...2...+...3...+...4...+...5...+...6...+...7...</pre> <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 Network Generation*, SH31-0131
- *TPF C/C++ Language Support User's Guide*, SH31-0121
- *TPF Concepts and Structures*, GH31-0139
- *TPF Database Reference*, SH31-0143
- *TPF General Macros*, SH31-0152
- *TPF Operations*, SH31-0162
- *TPF System Generation*, SH31-0171
- *TPF System Installation Support Reference*, SH31-0149
- *TPF System Macros*, SH31-0151

### IBM Transmission Control Protocol/Internet Protocol (TCP/IP) Books

- *International Technical Support Centers TCP/IP Tutorial and Technical Overview*, GG24-3376.
- *TCP/IP for MVS: Offloading TCP/IP Processing, Version 3, Release 1*, SC31-7133, See this book for information about installing the 3172 Model 3 Interconnect Controller with the offload program and related diagnostics.
- *TCP/IP Version 2 Release 2.1 for MVS: Programmer's Reference*, SC31-6087.
- *Transmission Control Protocol/Internet Protocol Version 2.0 for OS/2: Programmer's Reference*, SC31-6077.

### IBM Common Link Access to Workstation (CLAW) Books

- *AIX Block Multiplexor Channel Adapter User's Guide and Service Information*, SC23-2427.

### IBM 3172 Interconnect Controller Books

- *3172 Interconnect Controller Program, Version 3.3 User's Guide*, SC30-3572.

## Online Information

- *Messages (Online)*
- *Messages (System Error and Offline)*
- *SSL for the TPF 4.1 System: An Online 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

---

## Part 1. TCP/IP Support Overview

<b>Networking Protocols Introduction</b> . . . . .	3
TPF System Support of Transmission Control Protocol/Internet Protocol . . . . .	3
TCP/IP Native Stack Support . . . . .	4
<b>TCP/IP Network Overview</b> . . . . .	7
Network Requirements . . . . .	7
Industry-Standard Transport Application Programming Interface (API) . . . . .	7
Open Network Connectivity . . . . .	7
Client/Server Environment . . . . .	7
Enhanced Role in the Internet . . . . .	7
Porting Socket Applications . . . . .	7



---

## Networking Protocols Introduction

When networking protocols were being developed in the 1960s through the 1980s, expensive, dedicated hardware was required to transfer data over the network. Both protocol and hardware were proprietary and required special skills and tools to perform problem determination.

Today, this has changed dramatically with the standard protocol and technology that emerged during the early 1980s. This new technology, called internetworking, or internetting, accommodates multiple, diverse underlying hardware technologies by adding both physical connections and a new set of conventions. Internet technology hides the details of network hardware and permits computers to communicate independently of their physical network connections. One example of this is Transmission Control Protocol/Internet Protocol, also known as TCP/IP.

---

### TPF System Support of Transmission Control Protocol/Internet Protocol

Transmission Control Protocol/Internet Protocol (TCP/IP) supports an interconnection of computer networks that provides universal communication services.

A *computer network* is a group of connected nodes used for data communication. A computer network configuration contains:

- Data processing devices
- Software
- Transmission media linked for information interchange.

Figure 1 on page 4 shows how the TPF system, running on a host processor and using an IBM 3172 Model 3 Interconnect Controller has access to local area networks (LANs), a wide area network (WAN), and remote applications.

A *local area network* (LAN) is a computer network that connects computer systems that are in a limited geographical area. A LAN has distance limitations.

A *wide area network* (WAN) is a computer network that is usually in different cities or even different countries and, therefore, provides communication services to a geographic area larger than that served by a local area network (LAN).

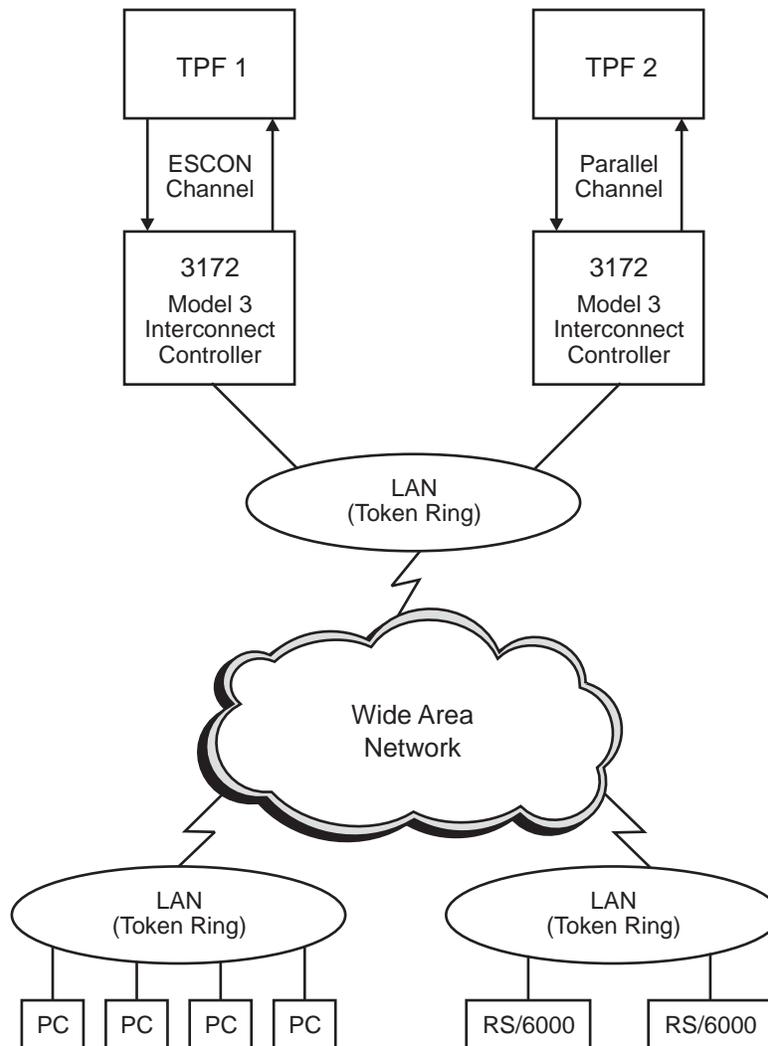


Figure 1. TCP/IP Sample Computer Network

## TCP/IP Native Stack Support

The original TCP/IP support required an offload device (IBM 3172). That support enabled TPF applications to use the socket application programming interface (API) to talk to remote socket applications. The socket calls were passed to the 3172 by using the Common Link Access to Workstation (CLAW) protocol. The offload server code in the 3172 would then run the socket call. In this environment, a thin socket layer existed in the TPF system. The full socket layer TCP, UDP, and IP layers existed in the 3172. Later, support was added to allow TPF to connect to another offload device, specifically the Cisco 7500 router.

With TCP/IP native stack support, the stack is incorporated in the TPF system itself. The IP over channel data link control (CDLC) link layer is included with this support, enabling TPF to connect to more IP router boxes in addition to continuing to support offload devices (see Figure 2 on page 5).

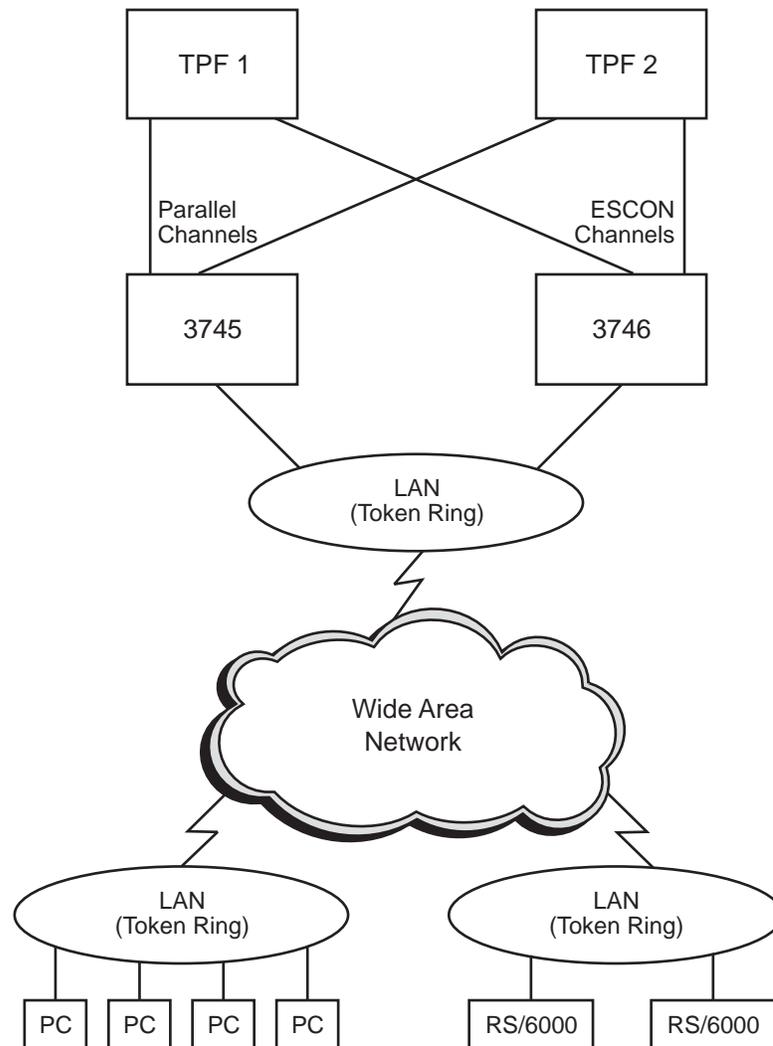


Figure 2. TCP/IP Native Stack Support Sample Computer Network Using Channel Data Link Control (CDLC)

With the introduction of an integrated hardware feature called the Open Systems Adapter (OSA) card, direct connectivity between IBM System/390 applications and remote TCP/IP applications is provided. Now, with the third generation OSA-Express card, OSA-Express support is enabled on the TPF system, allowing you to connect to high-bandwidth networks such as the Gigabit Ethernet (GbE or GENET) or Fast Ethernet (FENET). Queued direct I/O (QDIO) protocol is used to communicate between the TPF system and the OSA-Express card, enabling memory to be shared and reducing I/O. See Figure 3 on page 6 for an example of TCP/IP native stack support and the OSA-Express interface. See Figure 4 on page 6 for an example of TCP/IP native stack support combining both CDLC and OSA-Express interfaces.

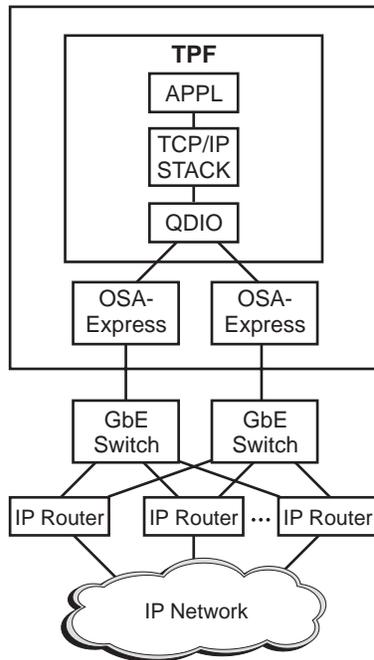


Figure 3. TCP/IP Native Stack Support Sample Computer Network Using the OSA-Express Interface

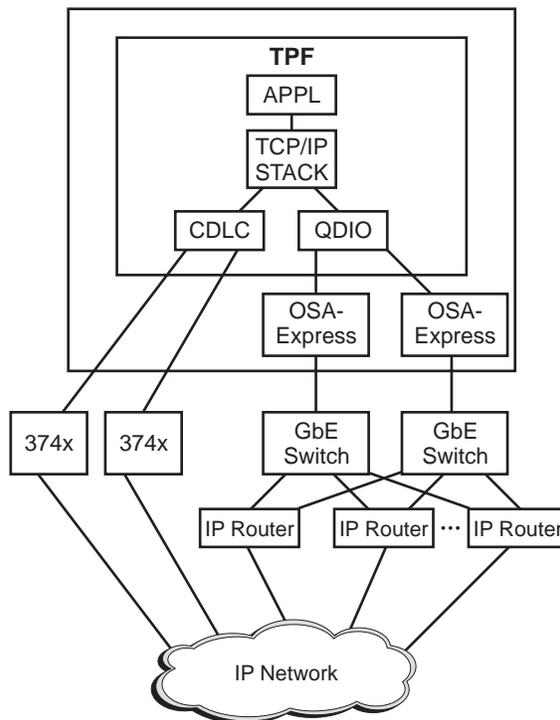


Figure 4. TCP/IP Native Stack Support Sample Computer Network Using Combined CDLC and OSA-Express Interfaces

---

# TCP/IP Network Overview

This chapter discusses customer requirements for a TCP/IP network.

---

## Network Requirements

As more and more applications are added to networks, we must consider customer requirements regarding these applications. The TPF implementation of Transmission Control Protocol/Internet Protocol (TCP/IP) meets customers requirements in the following areas:

- Industry-standard transport application programming interface (API)
- Open network connectivity
- Client/server environment
- Enhanced role in the Internet
- Porting socket applications.

## Industry-Standard Transport Application Programming Interface (API)

TCP/IP support provides a set of ISO-C functions, called *socket APIs*, that TPF applications use to access the Internet to take advantage of TCP/IP.

## Open Network Connectivity

TCP/IP allows communication between IBM systems or non-IBM systems on the same computer network or on other computer networks. TCP/IP is not vendor-specific and can be implemented on everything from the smallest personal computer to the largest mainframe.

## Client/Server Environment

User-written ISO-C socket applications can run in the TPF system to provide access to TPF data. These socket applications can be a client, a server, or both depending on whether they request or provide a service.

## Enhanced Role in the Internet

By using the transport and network interfaces in the offload device or IP router, the TPF system can receive and send data from remote socket applications.

## Porting Socket Applications

TCP/IP follows industry standards for all the socket ISO-C application programming interface (API) functions. This allows the porting of any application program that follows the industry standard for using socket interfaces.



---

## Part 2. TCP/IP Offload Support

<b>TCP/IP Internals</b> . . . . .	11
IBM 3172 Model 3 Interconnect Controller Overview . . . . .	11
Configuration Characteristics of TCP/IP Offload Devices. . . . .	12
Data Flow between the Offload Device and the TPF System . . . . .	12
Sample Transmission Control Protocol/Internet Protocol Network . . . . .	13
Components of TCP/IP Support. . . . .	16
Outbound Message Flow through the Socket/CLAW Interfaces . . . . .	17
Inbound Message Flow through the Socket/CLAW Interfaces . . . . .	18
Nonsocket User Exits . . . . .	20
Nonsocket Activation. . . . .	20
Nonsocket Connect . . . . .	20
Nonsocket Deactivation. . . . .	20
Nonsocket Message . . . . .	21
<b>TPF Control Block Structures</b> . . . . .	23
CLAW Device Table (CDT) and Related Control Block Structures . . . . .	23
Contents of the CLAW Device Table . . . . .	23
Related Control Block Structures . . . . .	23
Determining the Value for the CLAWADP Parameter . . . . .	23
File Descriptor Table and Related Control Block Structures. . . . .	24
Contents of the File Descriptor Table . . . . .	24
Related Control Block Structures . . . . .	24
CLAW Send Message Block . . . . .	24
Socket Thread Control Blocks . . . . .	24
Determining the Value for the CLAWFD Parameter . . . . .	25
Internet Protocol Address Table . . . . .	25
Contents of the Internet Protocol Address Table . . . . .	26
Related Control Block Structures . . . . .	26
Determining the Value for the CLAWIP Parameter . . . . .	26
Maximum Value for the IP Parameter. . . . .	26
Storage Considerations . . . . .	26
SNAKEY Parameters . . . . .	27
Miscellaneous Control Block Structures . . . . .	27
Socket Thread Control Blocks . . . . .	27
Calculating the Approximate Total Number of TCP/IP Bytes . . . . .	27
Example Calculating the Approximate Total Number of TCP/IP Bytes . . . . .	27
Tuning and Performance . . . . .	28
<b>Operator Procedures for TCP/IP Offload Support</b> . . . . .	29
Configuring a TCP/IP System . . . . .	29
Defining the CLAW Host Name . . . . .	29
Considerations for IBM 3172 Model 3 Interconnect Controllers . . . . .	29
Defining CLAW Workstations for a TPF Host Processor . . . . .	29
Considerations for IBM 3172 Model 3 Interconnect Controllers . . . . .	30
Activating and Deactivating CLAW Workstations. . . . .	30
Displaying Information about TCP/IP Support. . . . .	31
Deleting a CLAW Workstation . . . . .	31
Moving a CLAW Workstation from One TPF Host Processor to Another . . . . .	31
Performing a Hardware Switchover . . . . .	32
Using the CLAW Data Trace Function . . . . .	32
Starting the CLAW Data Trace Function. . . . .	32
Stopping the CLAW Data Trace Function . . . . .	32
Using the CLAW Process Trace Function . . . . .	33

Starting the CLAW Process Trace Function . . . . .	33
Stopping the CLAW Process Trace Function . . . . .	33
Resetting the ZCLAW Command Lock . . . . .	33

---

## TCP/IP Internals

This chapter provides:

- An overview of the IBM 3172 Model 3 Interconnect Controller
- Configuration characteristics of TCP/IP offload devices
- Data flow between the offload device and the TPF system
- An overview of the TCP/IP offload support components
- A description of the inbound and outbound message flow
- A description of the nonsocket user exits.

---

### IBM 3172 Model 3 Interconnect Controller Overview

The 3172 Model 3 Interconnect Controller with the Offload program, or a similar workstation, is integral to TCP/IP support. The 3172 Model 3 Interconnect Controller is configured as an offload device between the TPF host and local area networks (LANs).

Figure 5 on page 12 shows two 3172 Model 3 Interconnect Controllers with the Offload program defined to a TPF system. The following naming conventions are used:

- The socket application programming interface (API) support program in the TPF host has the TPF *host application name* of TCP/IP. This name is predefined by the 3172 Model 3. The socket API support program in the TPF host processor packages the socket API requests and sends them to the 3172 Model 3 Interconnect Controller for processing.
- The socket offload program in the 3172 Model 3 has the *workstation application (WSA) name* of API. This name is predefined by the 3172 Model 3. It processes the socket API requests sent from the TPF host. When communicating with the TPF host, it uses Common Link Access to Workstation (CLAW) protocol.
- The *symbolic device address (SDA)* represents a pair of subchannels connecting the TPF host processor to the TCP/IP offload device. The even-numbered subchannels are the *read* subchannels, while the odd-numbered subchannels are the *write* subchannels.
- If there is only one workstation, the name can be OS2TCP. If there is more than one workstation the additional names must be OS3172xx where xx is 2 unique alphabetical characters. In this figure, the names OS317201 and OS31702 are used.

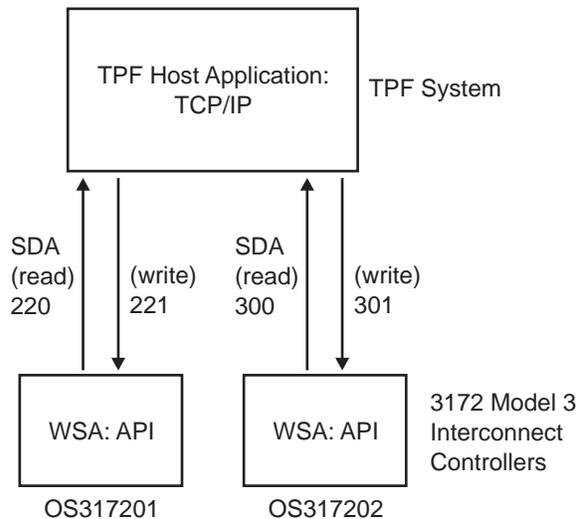


Figure 5. Two IBM 3172 Model 3 Interconnect Controller Offload Devices Connected to a Host Transaction Processing Facility Processor

---

## Configuration Characteristics of TCP/IP Offload Devices

TCP/IP offload devices have the following configuration characteristics:

- Each TCP/IP offload device is connected to the host processor through a pair of subchannels. The even-numbered subchannel on the host side is the *read* subchannel, and the odd-numbered subchannel of the even-odd subchannel pair is the *write* subchannel.
- You can connect as many as 84 TCP/IP offload devices to each TPF host processor.
- TCP/IP offload devices provide effective use of subchannels by executing Common Link Access to Workstation (CLAW) commands only when there is data to transfer between the host and the TCP/IP offload device.

---

## Data Flow between the Offload Device and the TPF System

The TPF system implementation of TCP/IP support is based on the IBM 3172 Model 3 Interconnect Controller with the TCP/IP Offload program and supports applications using the Berkeley Software Distribution (BSD) standard socket API functions to communicate with the Internet.

See “Socket Overview” on page 119 for a discussion of socket programming concepts used by TCP/IP support. Figure 6 on page 13 shows the data flow between the TPF system and the IBM 3172 Model 3 Interconnect Controller. The following steps correspond to numbers in Figure 6:

1. An application in the TPF system issues a socket API function and the socket API is forwarded to the TPF socket API support component.
2. The TPF socket API support component builds an inter-user communication vehicle (IUCV) message that contains the socket API function call, parameters, and user data. This message is transmitted through the TPF CLAW device interface to the IBM 3172 Model 3 Interconnect Controller.
3. The return code from the IBM 3172 Model 3 Interconnect Controller is returned to the application.

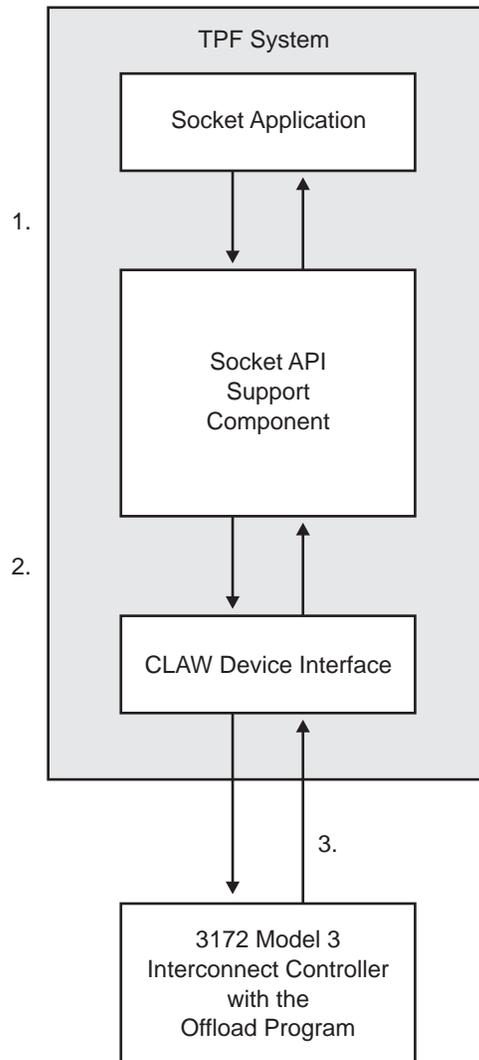


Figure 6. TPF Implementation of the IBM 3172 Model 3 Interconnect Controller with the Offload Program

---

## Sample Transmission Control Protocol/Internet Protocol Network

TCP/IP builds an interconnection of networks that provides universal communication services.

The service performed by TCP/IP support is transparent to the application and is comparable to services performed from other platforms.

An application can be either a client, server, or both, depending on whether the application requests or provides a service. Program TPF servers with special consideration to reduce resource usage and to handle messages most effectively. Figure 7 on page 15 shows the TPF system as the host with two IBM 3172 Model 3 offload devices.

The following steps correspond to the numbers in Figure 7 on page 15:

1. The TPF *host* is connected to both token-ring and fiber distributed data interface (FDDI) networks through the IBM 3172 Model 3 Interconnect Controller with the Offload program, and is able to forward or receive data from clients on any connected network.
2. *Parallel* or Enterprise Systems Connection (ESCON) channels connect the TPF host to the IBM 3172 Model 3 Interconnect Controller.
3. Each TPF host can connect to more than one *IBM 3172 Model 3 Interconnect Controller*.

The IBM 3172 Model 3 Interconnect Controller with the Offload program allows the TPF system to communicate with remote TCP/IP hosts through socket application program interface (API) functions. See “Socket Overview” on page 119 for a detailed description of sockets.

4. The 3172 Model 3 Interconnect Controllers send information *packets* between local area networks (LANs) and the TPF host processor. IBM recommends that all 3172 Model 3 Interconnect Controllers are connected to the same local area network (LAN). The 3172 Model 3 Interconnect Controller can connect to the following LANs:
  - Fiber distributed data interface (FDDI)
  - Ethernet
  - Token-ring.
5. *Routers* distribute information packets of data directly when the destination and source are on the same TCP/IP network or indirectly when the destination and source are on two different networks.
6. The information packets are routed to the Internet.

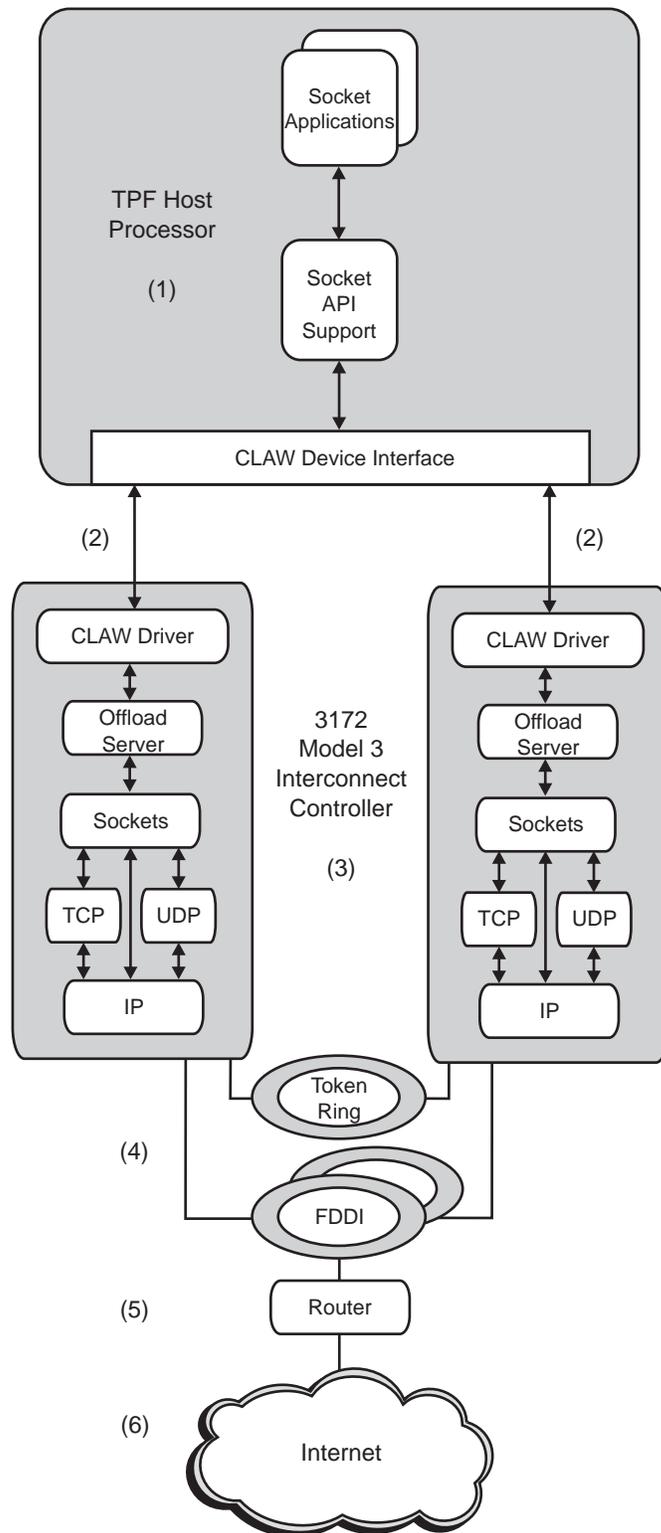


Figure 7. Network Overview

## Components of TCP/IP Support

Figure 8 provides an overview of the TCP/IP support components using the socket/CLAW interfaces.

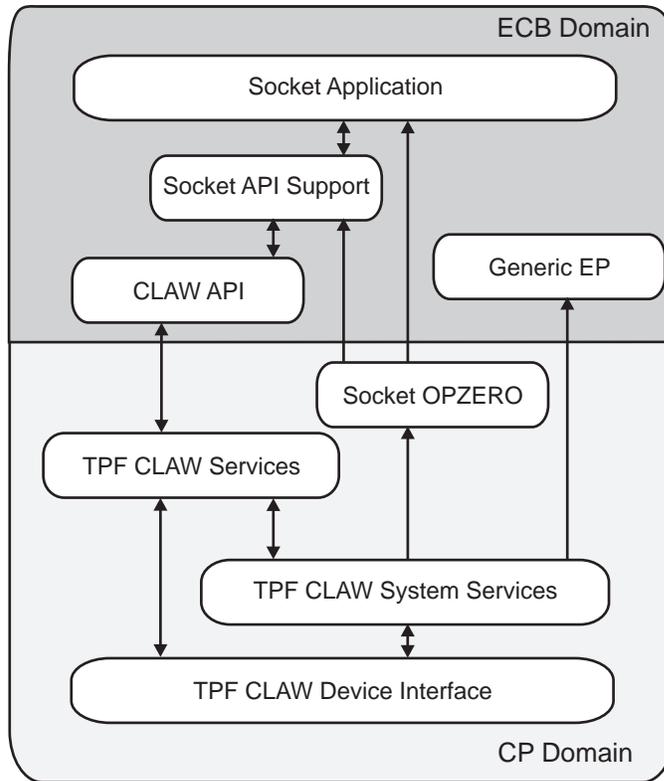


Figure 8. TCP/IP Support Components Overview

In Figure 8, the TCP/IP support components send socket API functions to the TCP/IP offload device to complete the processing of the functions and return responses for the functions to the TPF system. The TCP/IP support components that interface with the TCP/IP offload device are as follows:

### Socket application

Issues the standard ISO-C interface functions, called socket API functions, which enable data to be sent and received across the Internet.

### Socket API support

Provides the socket API functions and issues the CLAW API function needed to send the socket functions to the TCP/IP offload device. See “Socket Application Programming Interface Functions Reference” on page 137 for a description of the socket API functions.

### CLAW API

Provides restricted ISO-C interface functions to communicate with any CLAW workstation, such as the TCP/IP offload device. See Appendix E, “TCP/IP Restricted CLAW C Functions: Reference” on page 385 for a description of these functions.

### TPF CLAW services

Provides the control program service routines for the CLAW API functions and enters the TPF CLAW device interface to complete the processing of the CLAW functions.

**TPF CLAW system services**

Provides system services to the TPF CLAW device interface and TPF CLAW services including, control block management, message dispatching, and post-interrupt handling.

**TPF CLAW device interface**

Provides CLAW I/O functions, manages I/O queues, and handles I/O completions using the Common Link Access to Workstation (CLAW) protocol.

**Socket OPZERO**

A post-interrupt routine that receives inbound messages from TPF CLAW system services and forwards the messages to the socket application or to socket API support by creating new entry control blocks (ECB)s or posting ECBs that have been suspended by socket API support.

**Generic EP**

Provides entry points for receiving CLAW messages and enters the appropriate entry point program to display information regarding the message on the TPF console.

---

## Outbound Message Flow through the Socket/CLAW Interfaces

When a socket application issues a socket API function, socket API support determines whether the function needs to be sent to the TCP/IP offload device to complete the processing of the function. If the function needs to be sent to the TCP/IP offload device, socket API support issues a `claw_send` function call to send the socket API call to the Common Link Access to Workstation (CLAW) API. After CLAW API and TPF CLAW services have processed the `claw_send` function call, the TPF CLAW device interface issues the I/O function needed to send the socket API call to the TCP/IP offload device. See “`claw_send` — Send a Message on an Active Logical Link” on page 403 for a description of the `claw_send` function.

Figure 9 on page 18 shows the outbound message flow through the TCP/IP support components of a socket API function that is being sent to the TCP/IP offload device.

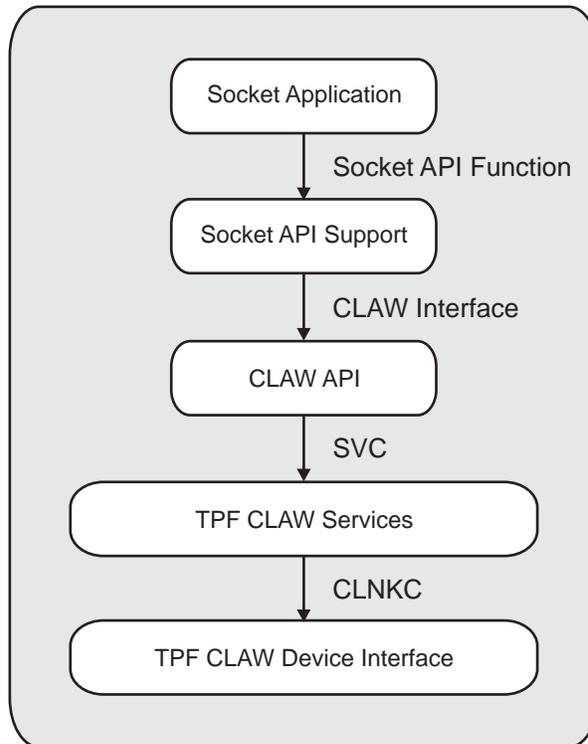


Figure 9. Outbound Message Flow

The following describes the outbound message flow for a socket API function that is being sent to the TCP/IP offload device:

- A socket application program issues a socket API function.
- Socket API support issues a `claw_send` function call to send the socket API call to the CLAW API.
- The CLAW API issues a supervisor call (SVC) to enter TPF CLAW services and to process the `claw_send` function call.
- TPF CLAW services processes the `claw_send` function call and issues a CLNKC macro to go to the TPF CLAW device interface.
- The TPF CLAW device interface manages the I/O queues and builds the channel program needed to send the socket API call to the TCP/IP offload device.

---

## Inbound Message Flow through the Socket/CLAW Interfaces

TPF Common Link Access to Workstation (CLAW) system services enables the TPF CLAW device interface to receive inbound messages from the TCP/IP offload device by sending an internal RECEIVE request to the TPF CLAW device interface. When TPF CLAW system services receives a message from the TCP/IP offload device, TPF CLAW system services forwards the message to TPF CLAW system services. TPF CLAW system services determines whether the message is to be forwarded to socket OPZERO or to the generic entry point (EP). If TPF CLAW system services forwards the message to socket OPZERO, socket OPZERO returns the message to the socket application either directly or through socket API support. If TPF CLAW system services forwards the message to the generic EP, generic EP enters the appropriate CLAW EP program to display information regarding the message on the TPF console.

Figure 10 shows an inbound message flow through the TCP/IP support components from the TCP/IP offload device.

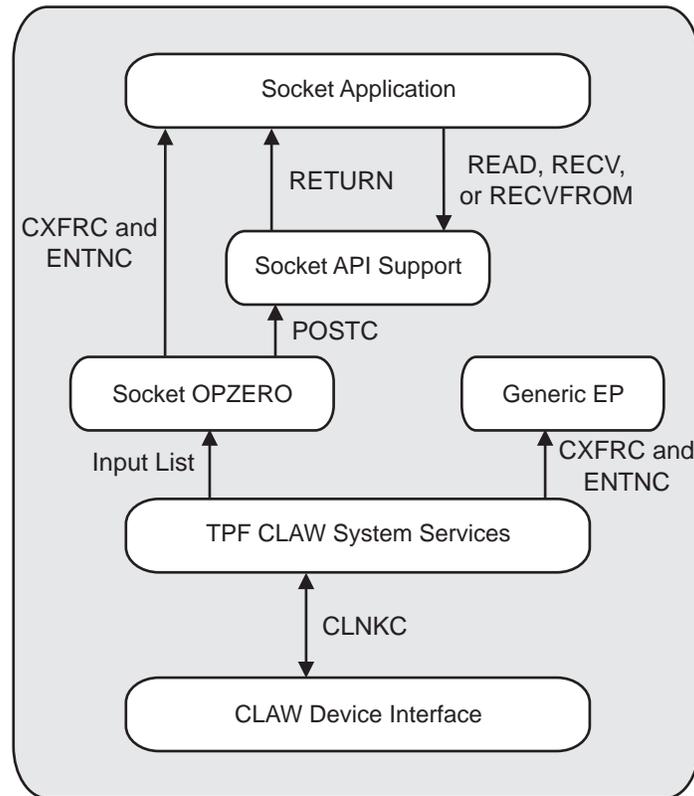


Figure 10. Inbound Message Flow

The following describes the inbound message flow for a message received from the TCP/IP offload device:

- TPF CLAW system services issues an internal RECEIVE request to the TPF CLAW device interface.
- When the TPF CLAW device interface receives an inbound message from the TCP/IP offload device, the TPF CLAW device interface issues a CLNKC macro to TPF CLAW system services to begin processing the message.
- TPF CLAW system services determines whether the message is to be forwarded to socket OPZERO or the generic entry point (EP). If TPF CLAW system services determines that the message is to be forwarded to socket OPZERO, it schedules the socket OPZERO post-interrupt routine on the input list to continue processing the message.
- If TPF CLAW system services determines that the message is to be forwarded to the generic EP, it issues the CXFRC and ENTNC macros to enter that component.

The generic EP forwards the unsolicited message to the appropriate CLAW EP program, which displays information regarding the message on the TPF console. The following CLAW EP programs are entered from the generic EP.

- The CLAW device failure EP is entered when the CLAW device interface detects an unrecoverable error and must shut down a subchannel pair to a CLAW workstation (TCP/IP offload device). See “claw\_openadapter — Initialize an Adapter” on page 397 for additional information on this CLAW EP.

- The CLAW DISCONNECT EP is entered when a DISCONNECT request is issued from a CLAW workstation to the host TPF system. See “claw\_accept — Accept a CONNECT Request from the Workstation” on page 386 or “claw\_connect — Initiate a Request to Open a Logical Link” on page 390 for additional information on this CLAW EP.
- The CLAW CONNECT EP is entered when a CLAW workstation initiates a CONNECT request on the workstation’s subchannel pair. This CLAW EP issues a claw\_accept request to complete the connection. See “claw\_openadapter — Initialize an Adapter” on page 397 for additional information on this CLAW EP.
- Socket OPZERO issues the CXFRC and ENTNC macros to create an entry control block (ECB) and enter a socket application if a socket message was received and an ECB was not waiting for the message. If the socket application is entered in this manner, an activate\_on\_receipt function call had been previously issued by a socket application.
- To complete the processing of the message and to receive the message, the socket application issues a read, recv, or recvfrom socket API function call to socket API support. See “activate\_on\_receipt — Activate a Program after Data Received” on page 144 for a description of the activate\_on\_receipt socket function.
- Socket OPZERO issues a POSTC macro if a socket message was received and an ECB was waiting for the message. The POSTC macro enters socket API support by activating the ECB that had been previously suspended by socket API support.
- Socket API support returns the message to the socket application program that had been waiting for the message. When the socket application is entered this way, the processing of the message is completed.

---

## Nonsocket User Exits

TCP/IP support provides the following nonsocket user exits:

- Nonsocket activation
- Nonsocket connect
- Nonsocket message
- Nonsocket deactivation.

## Nonsocket Activation

The nonsocket activation user exit is entered for the following conditions:

- When CLAW workstations are being connected during system cycle-up and the CLAW workstation application name is not API
- During ZCLAW ACTIVATE processing when the workstation application name of the CLAW workstation being activated is not API.

## Nonsocket Connect

The nonsocket connect user exit is activated during CLAW connect processing and allows nonsocket Common Link Access to Workstation (CLAW) applications to be activated.

## Nonsocket Deactivation

The nonsocket deactivation user exit is entered for the following reasons:

- When CLAW workstations are being disconnected during system cycle-down and the CLAW workstation application name is not API
- During ZCLAW INACTIVATE processing when the workstation application name of the CLAW workstation being deactivated is not API.

## **Nonsocket Message**

Socket OPZERO enters the nonsocket message user exit when a nonsocket message arrives from the TCP/IP offload device.

See *TPF System Installation Support Reference* for additional information about the nonsocket user exits. See “Socket Overview” on page 119 and the *TPF System Installation Support Reference* for information about the socket user exits.



---

## TPF Control Block Structures

This chapter discusses the following:

- The control block structures used by TCP/IP support.
- The CLAWADP, CLAWFD, and CLAWIP SNAKEY parameters. See *TPF ACF/SNA Network Generation* for information about coding these parameters (and the SOCKSWP parameter) in the SNAKEY macro.
- Storage considerations for TCP/IP support.
- Tuning and performance.

---

## CLAW Device Table (CDT) and Related Control Block Structures

The Common Link Access to Workstation (CLAW) device table is a TCP/IP control block structure located in main storage that contains information about each CLAW workstation defined in the TPF system. The CDT is also saved on file to preserve changes made to the core copy of the CDT across an initial program load (IPL). Whenever a CLAW workstation is added to the TPF system using a ZCLAW ADD command, an entry is added to the CLAW device table.

See *TPF Operations* for more information about the ZCLAW ADD command and for restrictions on its use.

### Contents of the CLAW Device Table

The CLAW device table contains information such as adapter ID, workstation name, workstation application, host application, symbolic device address (SDA), and device status.

An entry is deleted from the CDT by using the ZCLAW DELETE command.

Each TPF host processor in the TPF system has its own unique CDT. Therefore, the CDT contents are not necessarily consistent across all of the processors in a loosely coupled TPF system.

See *TPF Operations* for more information about the ZCLAW DELETE command and for restrictions on its use.

### Related Control Block Structures

There are many CLAW control block structures associated with the CDT. These control blocks are obtained by the TCP/IP support during CLAW I/O processing and returned to the system at various times during the I/O processing using a `claw_closeadapter` feature. See “`claw_closeadapter` — Terminate CLAW Activity on Subchannel Pair” on page 388 for a description of the `claw_closeadapter` function.

---

## Determining the Value for the CLAWADP Parameter

The number of CLAW device entries and associated CLAW control blocks are defined using the CLAWADP parameter of the SNAKEY macro. In choosing a CLAWADP value for a particular processor use the following:

CLAWADP = Number TCP/IP offload devices attached to a processor

**Note:** The maximum value for CLAWADP is 84 because a maximum of 84 CDT entries can be saved on file for a particular processor.

For example, if you plan to attach 10 IBM 3172 Model 3 Interconnect Controllers to processor A, set CLAWADP for that processor to 10. When defining CLAWADP for a processor, do not set the value larger than the number of TCP/IP offload devices you will be attaching to that processor because the amount of storage needed for the CLAW control blocks increases significantly as the CLAWADP value is increased. See “Storage Considerations” on page 26 for a discussion of storage.

---

## File Descriptor Table and Related Control Block Structures

The file descriptor table (FDT) is a TCP/IP control block structure located in main storage that contains information about each file descriptor defined in the TPF system. A file descriptor, which contains socket status information, is obtained by TCP/IP support when a socket application issues a socket or accept function call. For every socket descriptor returned to the socket application with a function call, such as the socket or accept function call, there is one file descriptor entry associated with the socket descriptor integer value. In addition, each file descriptor is associated with one TCP/IP offload device.

### Contents of the File Descriptor Table

The file descriptor table contains information such as local socket descriptor, remote socket description, adapter ID, path ID, address domain, socket type, protocol, and socket status.

A file descriptor is returned to the system when:

- A socket application issues a close of its associated local socket descriptor.
- The TCP/IP offload device associated with the file descriptor is disconnected from the TPF system.
- A system error occurs, and the ECB that obtained the file descriptor exits.

Each processor in the TPF system has its own unique FDT. Therefore, the FDT contents are not necessarily consistent across all of the processors in a loosely coupled TPF system.

### Related Control Block Structures

The following control blocks are related to the file descriptor table:

- CLAW send message block
- Socket thread control blocks.

#### CLAW Send Message Block

The CLAW send message block (ICMSGB) is obtained during the processing of a `c1aw_send` function call. A `c1aw_send` function call is issued by TCP/IP support when a socket API function call will be processed by the TCP/IP offload device. Each ICMSGB used is returned to the system when `c1aw_send` function call has been completed. See “`c1aw_send` — Send a Message on an Active Logical Link” on page 403 for a description of the `c1aw_send` function.

#### Socket Thread Control Blocks

Different types of socket thread control blocks are chained off individual file descriptors. A type represents a unique function, such as the accept function to accept a connection request, or the connect function to request a connection to a remote host.

Most of the events associated with each of the offload devices are thread control blocks chained off any one file descriptor. The different types of socket thread control blocks that can be chained off a file descriptor are:

- accept
- read, or recv, or recvfrom
- send, or sendto, or write, or writev, chained only when they are written to a datagram socket.
- connect
- shutdown
- close
- listen
- bind or socket
- gethostid
- gethostname
- getpeername
- getsockname
- getsockopt
- ioctl
- setsockopt.

---

## Determining the Value for the CLAWFD Parameter

The number of file descriptor entries and CLAW send message blocks is defined using the CLAWFD parameter of the SNAKEY macro. The value of the CLAWFD parameter for a particular processor depends on the CLAWADP value you are defining for that processor and the number of file descriptors you define for each TCP/IP offload device. Increasing the CLAWFD value does not significantly raise the amount of main storage needed, so set the number of file descriptors you define for each TCP/IP offload device equal to the maximum number of socket descriptors which can be created for each TCP/IP offload device. The maximum number of socket descriptors which can be created for the IBM 3172 Model 3 Interconnect Controller is 2000. In choosing a CLAWFD value for a particular processor, use the following:

CLAWFD = Number of file descriptors × number  
of TCP/IP offload devices attached to a processor

**Note:** To avoid the depletion of file descriptors when you are running your socket applications, set the number of file descriptors to the maximum value.

**Example:** If you plan to use three IBM 3172 Model 3 Interconnect Controllers on processor A, set the CLAWFD parameter for that processor as follows:

$2000 \times 3 = 6000$

---

## Internet Protocol Address Table

The Internet Protocol address table (IPT) is a TCP/IP control block structure that contains information about each TCP/IP offload device interface address and is located in main storage. Each TCP/IP offload device may define one or more interfaces with the Internet or IP addresses. One or more IPT entries are obtained whenever TCP/IP support connects a TCP/IP offload device to the TPF system. TCP/IP support saves each address in the IPT so that it can be obtained when a socket application issues a bind API function call. When a socket application issues

a bind for a specific IP address, TCP/IP support compares the IP address coded by the application with the one saved in the IPT to ensure that it is valid.

## Contents of the Internet Protocol Address Table

Each Internet Protocol (IP) entry contains information such as IP address, adapter ID, and path ID.

An IP entry is returned to the system when its associated TCP/IP offload device is disconnected from the system.

Each processor in the TPF system has its own unique IPT. Therefore, the IPT contents are not necessarily consistent across all of the processors in a loosely coupled TPF system.

## Related Control Block Structures

There are no related control block structures for the Internet Protocol address table (IPT).

---

## Determining the Value for the CLAWIP Parameter

The number of IP entries is defined using the CLAWIP parameter of the SNAKEY macro. Like the CLAWFD value, increasing the CLAWIP value does not significantly raise the amount of main storage needed. In choosing a CLAWIP value for a particular processor, use the following:

CLAWIP = Total number of IP addresses for each TCP/IP offload device attached to a processor

## Maximum Value for the IP Parameter

The maximum value that you can define for the IP parameter is:

The sum of 81 + 81 + ...  
up to 84 TCP/IP offload devices attached to processor = 6804

where:

- The maximum number of IP addresses that can be defined on one TCP/IP offload device is 81.
- The maximum number of TCP/IP offload devices that can be attached per processor is 84. See "Determining the Value for the CLAWADP Parameter" on page 23.

**Example:** For example, if you are attaching TCP/IP offload device 1 with 3 IP addresses and TCP/IP offload device 2 with 2 IP addresses to processor B, set CLAWIP to 5 for processor B.

$3 + 2 = 5$

---

## Storage Considerations

To calculate the number of bytes you will need for TCP/IP control block structures, you must consider:

- The control block structures identified in the SNAKEY macro
- Miscellaneous control blocks not associated with SNAKEY parameters
- Socket thread control blocks.

## SNAKEY Parameters

The following list shows the approximate number of bytes of storage allocated for TCP/IP control block structures for SNAKEY parameters:

Parameter	Approximate Number of Bytes
CLAWADP	303 220
CLAWFD	280
CLAWIP	64

## Miscellaneous Control Block Structures

There are approximately 4 120 bytes allocated for additional TCP/IP control block structures whose size does not depend on the values of any of the three SNAKEY values described in this chapter.

## Socket Thread Control Blocks

There are 32 bytes allocated for each socket thread control block. The number of socket thread control blocks allocated is equal to the total number of entry control blocks (ECBs) in the system.

## Calculating the Approximate Total Number of TCP/IP Bytes

The following formula calculates the approximate total number of bytes required for the TCP/IP control block structures:

$$(303\ 220 \times \text{clawadp}) + (280 \times \text{clawfd}) + (64 \times \text{clawip}) + \text{miscellaneous} + (32 \times \text{ecbs}) = \text{the total number of bytes}$$

where:

**clawadp**

is the value set for the CLAWADP parameter in the SNAKEY macro.

**clawfd**

is the value set for the CLAWFD parameter in the SNAKEY macro.

**clawip**

is the value set for the CLAWIP parameter in the SNAKEY macro.

**miscellaneous**

is 4120 bytes in block structures not associated with the SNAKEY macro.

**ecbs**

is the total number of ECBs in the system.

## Example Calculating the Approximate Total Number of TCP/IP Bytes

Table 1 assumes values for the given SNAKEY parameters. Based on these values, the table shows the total number of bytes required for TCP/IP.

Table 1. Assumed TCP/IP Values

Parameter	Approximate Number of Bytes for Each Value	Value Used	Total Number of Bytes for Each Parameter
CLAWADP	303 220	2	606 440
CLAWFD	280	4000	1 120 000
CLAWIP	64	3	192
ECBs	32	200	6 400

Table 1. Assumed TCP/IP Values (continued)

Parameter	Approximate Number of Bytes for Each Value	Value Used	Total Number of Bytes for Each Parameter
Miscellaneous	4 120	N/A	4 120
Total	N/A	N/A	1 737 152

The approximate total number of bytes allocated for the control block structures allocated by SNAKEY parameters, socket thread control blocks, and miscellaneous bytes required for TCP/IP support for this example is:

$$(303\ 220 \times 2) + (280 \times 4000) + (64 \times 3) \\ + (200 \times 32) + 4120 = 1\ 737\ 752 \text{ total bytes}$$

---

## Tuning and Performance

The values assigned to CLAWADP, CLAWFD, CLAWIP, and SOCKSWP do not affect the performance of the TCP/IP offload device. TCP/IP support does not have the ability to improve the performance of any TCP/IP offload device.

---

# Operator Procedures for TCP/IP Offload Support

This chapter describes the tasks that an operator can perform for TCP/IP support.

---

## Configuring a TCP/IP System

To use TCP/IP support on your TPF system you must:

1. Install the IBM 3172 Model 3 Interconnect Controller following the installation instructions in *TCP/IP for MVS: Offloading TCP/IP Processing, Version 3, Release 1*.
  2. Define the host name for the TPF host processor using the ZCLAW ADD command. See “Defining the CLAW Host Name”.
  3. Define the CLAW workstations using the ZCLAW ADD command. See “Defining CLAW Workstations for a TPF Host Processor”.
  4. Activate the CLAW workstation using the ZCLAW ACTIVATE command. See “Activating and Deactivating CLAW Workstations” on page 30.
- 

## Defining the CLAW Host Name

When you install TCP/IP support on a TPF host processor, you must define the Common Link Access to Workstation (CLAW) host name for that processor. The *CLAW host name* is the name assigned to a TPF host processor that is used by a CLAW workstation to identify that TPF host processor.

To define the CLAW host name for a TPF host processor, enter the ZCLAW ADD command with the HOSTNAME parameter specified. See *TPF Operations* for more information about the ZCLAW ADD command.

## Considerations for IBM 3172 Model 3 Interconnect Controllers

If you are using an IBM 3172 Model 3 Interconnect Controller for your TCP/IP offload device, the CLAW host name for the TPF host processor must be TCP/IP. Otherwise, you cannot activate the TCP/IP offload device.

---

## Defining CLAW Workstations for a TPF Host Processor

When you IPL a TPF host processor, that supports TCP/IP, you must define the Common Link Access to Workstation (CLAW) workstation for that processor using the ZCLAW ADD command. As you define CLAW workstations, they are added to the CLAW device table (CDT) and filed out to fixed file. On subsequent IPLs the table is read back into storage and you do not have to reissue the ZCLAW command.

A *CLAW workstation* is a device that communicates with the TPF system using the Common Link Access to Workstation (CLAW) protocol. The IBM 3172 Model 3 Interconnect Controller is an example of a CLAW workstation.

A CLAW workstation definition consists of the following:

- Name of the CLAW workstation
- Symbolic device address (SDA) of the read channel unit for the CLAW workstation
- Name of the CLAW host application
- Name of the CLAW workstation application.

The *CLAW host application* is the application on the TPF host processor that is used to establish a CLAW connection with the CLAW workstation application.

The *CLAW workstation application* is the application on the CLAW workstation that is used to establish a CLAW connection with the CLAW host application.

To define a CLAW workstation, enter the ZCLAW ADD command. See *TPF Operations* for more information about the ZCLAW ADD command.

## Considerations for IBM 3172 Model 3 Interconnect Controllers

If you are using an IBM 3172 Model 3 Interconnect Controller for your TCP/IP offload device, you must specify the following values for the CLAW workstation definition:

### CLAW workstation name

Each workstation attached to the TPF host processor must have a unique name according to the following conventions:

- If there is only one workstation, the name of that workstation can be OS2TCP.
- If there is more than one workstation, the name of the additional workstations must have the format: OS3172xx, where xx is 2 unique alphanumeric characters (for example, OS317201).

### CLAW workstation application name

Must be API.

### CLAW host application name

Must be TCPIP.

---

## Activating and Deactivating CLAW Workstations

After you define the CLAW workstations for a TPF host processor, you must activate them. Activating a CLAW workstation allows socket applications on the TPF host processor to establish connections with other socket applications on a remote TCP/IP device when the TPF system is cycled to CRAS state or higher.

To activate a CLAW workstation, enter the ZCLAW ACTIVATE command. See *TPF Operations* for more information about the ZCLAW ACTIVATE command.

Use the ZCLAW INACTIVATE command to deactivate a CLAW workstation and the application sessions that were established through that CLAW workstation. See *TPF Operations* for more information about the ZCLAW INACTIVATE command.

**Note:** If the TPF system is cycled below CRAS state, the application sessions are automatically deactivated even though the CLAW workstation is still active.

If you perform an initial program load (IPL) while one or more CLAW workstations are active, the TPF system will automatically try to reactivate those CLAW workstations when it cycles to 1052 state. Socket connections can be established again when the TPF system cycles to CRAS state or higher.

---

## Displaying Information about TCP/IP Support

Use the ZCLAW DISPLAY command to display the following information:

- CLAW workstation definitions
- Status of a CLAW workstation (not active, active, or connected)
- List of the active CLAW workstations
- CLAW host name for the TPF host processor.

A CLAW workstation is *active* if you entered the ZCLAW ACTIVATE command for that CLAW workstation. In addition, a CLAW workstation is connected when the socket client has issued a CLAW connect to the offload application on the offload box.

Use the ZCLAW STATUS command to display the traffic load for a particular CLAW workstation. The *traffic load* is the number of bytes and messages sent and received for a CLAW workstation during a given period of time.

See *TPF Operations* for more information about the ZCLAW DISPLAY, ZCLAW STATUS, and ZCLAW ACTIVATE commands.

---

## Deleting a CLAW Workstation

You can delete a CLAW workstation if you no longer want to use it in your network configuration. To delete a CLAW workstation, do the following:

1. If the CLAW workstation is active, enter the ZCLAW INACTIVATE command to deactivate it.  
See *TPF Operations* for more information about the ZCLAW INACTIVATE command.
2. Enter the ZCLAW DELETE command to delete the CLAW workstation.  
See *TPF Operations* for more information about the ZCLAW DELETE command.

---

## Moving a CLAW Workstation from One TPF Host Processor to Another

If you are expanding or collapsing your loosely coupled TPF system and need to move a CLAW workstation from one TPF host processor to another TPF host processor, do the following:

1. From the TPF host processor where the CLAW workstation is currently connected, enter the ZCLAW INACTIVATE command to deactivate the CLAW workstation if the CLAW workstation is active.  
See “Activating and Deactivating CLAW Workstations” on page 30 for more information about deactivating a CLAW workstation. See *TPF Operations* for more information about the ZCLAW INACTIVATE command.
2. Physically connect the CLAW workstation to the new TPF host processor.
3. If you have not already defined the CLAW workstation to the new TPF host processor, enter the ZCLAW ADD command to define the CLAW workstation to the new TPF host processor.  
See “Defining CLAW Workstations for a TPF Host Processor” on page 29 for more information about defining a CLAW workstation. See *TPF Operations* for more information about the ZCLAW ADD command.
4. From the new TPF host processor, enter the ZCLAW ACTIVATE command to activate the CLAW workstation.

See “Activating and Deactivating CLAW Workstations” on page 30 for more information about activating a CLAW workstation. See *TPF Operations* for more information about the ZCLAW ACTIVATE command.

---

## Performing a Hardware Switchover

If you are performing a hardware switchover (that is, moving a TPF host processor to a new physical hardware device), use the same processor ID in the new physical hardware device for the TPF host processor.

If you use the same processor ID, you do not need to redefine the CLAW host name for the TPF host processor. In addition, if you use the same SDAs to connect the CLAW workstations to the new physical hardware device, you do not need to redefine the CLAW workstations to the TPF host processor, and the CLAW workstations will automatically be reactivated when the TPF system reaches 1052 state (if they were active before the hardware switchover was performed).

---

## Using the CLAW Data Trace Function

Use the CLAW data trace function to trace the messages that are sent and received between the TPF host processor and one or more CLAW workstations. This trace information can be useful when you are debugging your TPF socket applications because the messages that are traced include the requests *from* and the responses *to* the TPF socket applications.

See Part 4, “Socket Application Programming Interface Overview” on page 117 for more information about socket applications.

See Appendix A, “CLAW Trace Postprocessor” on page 343 for sample JCL as well as sample data trace output.

You can write the CLAW trace information to the real-time (RTA or RTL) tape and then use the CLAW trace postprocessor (CLTD) to format and print data trace information.

## Starting the CLAW Data Trace Function

To start the CLAW data trace function, do the following:

1. Enter the ZCLAW TRACE command with the START parameter specified. Also specify the WS parameter or the ALL parameter to indicate which CLAW workstations you want to trace.
2. Enter the ZCLAW TRACE command with the START TAPE parameters specified to start writing the CLAW data trace information to the real-time (RTA or RTL) tape.

See *TPF Operations* for more information about the ZCLAW TRACE command.

## Stopping the CLAW Data Trace Function

To stop the CLAW data trace function (and stop writing the trace information to tape), do the following:

1. Enter the ZCLAW TRACE command with the STOP ALL parameters specified.
2. Enter the ZTOFF command to remove the tape.

See *TPF Operations* for more information about the ZCLAW TRACE and ZTOFF commands.

---

## Using the CLAW Process Trace Function

Use the process trace function to trace system routines in the CCLAW1 CSECT. The process trace function is more of a system programming tool that you can use to debug any problems with the CLAW system process. Use the process trace function to debug CLAW APIs.

See Appendix E, “TCP/IP Restricted CLAW C Functions: Reference” on page 385 for more information about CLAW C functions.

See Appendix A, “CLAW Trace Postprocessor” on page 343 for sample JCL as well as sample process trace output.

You can write the CLAW process trace information to the real-time (RTA or RTL) tape and then use the CLAW process trace postprocessor (CLTP) to format and print the trace information.

## Starting the CLAW Process Trace Function

To start the CLAW process trace function, do the following:

1. Enter the ZCLAW TRACE command with the START parameter specified. Also specify the PROCESS parameter.
2. Enter the ZCLAW TRACE command with the START TAPE parameters specified to start writing the CLAW process trace information to the real-time (RTA or RTL) tape.

See *TPF Operations* for more information about the ZCLAW TRACE command.

## Stopping the CLAW Process Trace Function

To stop the CLAW process trace function (and stop writing the trace information to tape), do the following:

1. Enter the ZCLAW TRACE command with the STOP ALL parameters specified.
2. Enter the ZTOFF command to remove the tape.

See *TPF Operations* for more information about the ZCLAW TRACE and ZTOFF commands.

---

## Resetting the ZCLAW Command Lock

Each time you enter a ZCLAW command, a lock is set by the TPF system that prevents you from entering another ZCLAW command until processing is completed for the first ZCLAW command.

If the TPF system cannot complete processing for a ZCLAW command (that is, processing is hung), enter the ZCLAW RESET command to reset the ZCLAW command lock. This will allow you to enter other ZCLAW commands even though processing for the first ZCLAW command never completed.

**Attention:** Use the ZCLAW RESET command only in a test environment because the results cannot be predicted.

See *TPF Operations* for more information about the ZCLAW RESET command.



---

## Part 3. TCP/IP Native Stack Support

<b>TCP/IP Native Stack Support Internals</b> . . . . .	39
TCP/IP Layers . . . . .	39
Using CDLC IP Routers . . . . .	41
Configuration Characteristics of CDLC IP Routers . . . . .	41
Data Flow between the CDLC IP Router and the TPF System . . . . .	41
Sample TCP/IP Networks . . . . .	41
Using OSA-Express Support . . . . .	43
Configuration Characteristics of the OSA-Express Card . . . . .	43
Data Flow between the OSA-Express Card and the TPF System . . . . .	44
Components of TCP/IP Native Stack Support. . . . .	44
Policy Agent . . . . .	46
Outbound Message Flow . . . . .	46
Outbound Message Flow for CDLC . . . . .	47
Outbound Message Flow for OSA-Express . . . . .	48
Inbound Message Flow . . . . .	48
<b>TPF Control Block Structures</b> . . . . .	51
Socket Block Table Structure . . . . .	51
Defining the Socket Block Table . . . . .	51
CDLC IP Configuration Record . . . . .	51
Defining the CDLC IP Network Configuration . . . . .	52
CDLC IP CCW Area Table. . . . .	52
Defining CDLC IP CCW Area Table Resources . . . . .	52
OSA Configuration Record . . . . .	52
OSA Control Block Table . . . . .	52
OSA Shared IP Address Table . . . . .	52
OSA Read Buffers . . . . .	53
Defining OSA Read Buffers . . . . .	53
IP Message Table . . . . .	53
Defining the IP Message Table . . . . .	54
IP Routing Table . . . . .	54
Defining the IP Routing Table . . . . .	54
Tuning TCP/IP Native Stack Support . . . . .	54
Tuning Major Control Block Structures . . . . .	55
Tuning the IP over CDLC Link Layer . . . . .	55
Tuning the IP Network . . . . .	55
Performance. . . . .	56
TCP/IP Network Configurations . . . . .	56
Selecting the Local IP Address . . . . .	56
Choosing Network Paths . . . . .	57
Defining Gateways . . . . .	58
Choosing a Gateway. . . . .	58
<b>Operator Procedures for TCP/IP Native Stack Support</b> . . . . .	59
Configuring a TPF System . . . . .	59
Enabling TCP/IP Native Stack Support . . . . .	60
Local IP Addresses . . . . .	60
Default Local IP Address . . . . .	61
Maximum Packet Size . . . . .	61
Types of TPF Local IP Addresses . . . . .	62
CDLC Addresses . . . . .	62
Defining CDLC IP Addresses. . . . .	62
Deleting CDLC Local IP Addresses . . . . .	62

Restricting CDLC IP Addresses . . . . .	63
Real OSA IP Addresses . . . . .	63
Defining Real OSA IP Addresses . . . . .	63
Deleting Real OSA IP Addresses . . . . .	63
Virtual IP Addresses (VIPAs) . . . . .	63
Types of VIPAs . . . . .	64
Defining VIPAs to an OSA-Express Connection . . . . .	64
Displaying VIPAs . . . . .	65
Deleting VIPAs . . . . .	65
CDLC IP Connections . . . . .	65
Defining CDLC IP Connections . . . . .	65
Activating and Deactivating CDLC IP Routers . . . . .	65
Deleting CDLC IP Routers. . . . .	66
OSA-Express Connections . . . . .	66
Defining OSA-Express Cards to the Processor . . . . .	66
Defining OSA-Express Connections to TPF . . . . .	67
Activating and Deactivating OSA-Express Connections . . . . .	67
Displaying OSA-Express Connections . . . . .	67
Deleting OSA-Express Connections . . . . .	67
Gateways . . . . .	68
Routing Information Protocol . . . . .	68
How Network and Processor Failures Affect VIPAs. . . . .	68
Swinging VIPAs to an Alternate OSA-Express Connection . . . . .	68
Moving VIPAs from One Processor to Another . . . . .	68
Processor Deactivation . . . . .	69
The Operator and the ZVIPA Command. . . . .	69
Moving a VIPA by an Application Program . . . . .	69
Workload Balancing Using Movable VIPAs. . . . .	69
Configuration Examples . . . . .	71
Managing IP Routing Table Entries . . . . .	74
Displaying TCP/IP Native Stack Support . . . . .	75
Starting and Stopping the IP Trace Function . . . . .	75
Displaying IP Trace Information . . . . .	75
Using the Individual IP Trace Function . . . . .	76
Displaying Individual IP Trace Tables . . . . .	77
Deactivating Sockets. . . . .	77
Displaying Socket Control Block Information . . . . .	78
<b>Socket Application Design Considerations . . . . .</b>	<b>79</b>
Sharing Sockets . . . . .	79
Using Existing Socket Applications. . . . .	79
New Socket Options Supported. . . . .	80
Send Buffer and Receive Buffer Sizes . . . . .	80
Timeouts . . . . .	81
Low-Water Marks . . . . .	81
activate_on_accept API . . . . .	82
Local Sockets . . . . .	82
<b>Simple Network Management Protocol Agent Support . . . . .</b>	<b>85</b>
SNMP Overview . . . . .	85
SNMP Manager . . . . .	85
SNMP Agent. . . . .	85
Management Information Base (MIB). . . . .	85
Interaction between SNMP Components . . . . .	86
Protocol Data Units (PDUs) . . . . .	87
Structure and Fields of SNMP PDUs . . . . .	87

TPF SNMP Agent Support . . . . .	89
Implementing Management Information Base-II (MIB-II) . . . . .	90
Processing SNMP Requests . . . . .	91
Message Processing . . . . .	91
User MIB Variables . . . . .	93
SNMP Traps . . . . .	93
Installing TPF SNMP Agent Support . . . . .	96
Installing and Defining TPF TCP/IP Native Stack Support . . . . .	96
Installing the SNMP Agent . . . . .	96
Creating the SNMP Configuration File . . . . .	96
Coding the UCOM and UMIB User Exits . . . . .	98
Defining and Starting the SNMP Agent Server . . . . .	98
Defining IP Routing Table Entries . . . . .	99
Defining the TPF System to the SNMP Manager . . . . .	99
<b>Domain Name System Support . . . . .</b>	<b>101</b>
DNS Server . . . . .	101
TPF Host Name Table . . . . .	101
IP Address Selection . . . . .	102
DNS Client . . . . .	103
<b>Internet Security . . . . .</b>	<b>105</b>
Denial-of-Service Attacks . . . . .	105
Packet Filtering . . . . .	105
Packet Filtering Rules File Syntax . . . . .	106
Packet Filtering Default Rule . . . . .	107
Considerations for Packet Filtering Rules . . . . .	108
Order of Rules . . . . .	108
Performance Considerations . . . . .	108
Examples of Packet Filtering Rules . . . . .	108
Problem Diagnosis . . . . .	109
<b>TCP/IP Network Services Database Support . . . . .</b>	<b>111</b>
Quality of Service . . . . .	111
Data Collection and Reduction . . . . .	112
Message Counts by Application . . . . .	112
TCP/IP Network Services Database File . . . . .	113
TCP/IP Network Services Database File Syntax . . . . .	113
TCP/IP Network Services Database File Example . . . . .	114



# TCP/IP Native Stack Support Internals

This chapter provides:

- Configuration characteristics of IP routers
- Data flow between the IP router and the TPF system
- An overview of the TCP/IP native stack support components
- A description of the inbound and outbound message flow.

## TCP/IP Layers

Figure 11 shows the TCP/IP layers that reside in the TPF host and those that reside in the 3745 or 3746 IP router.

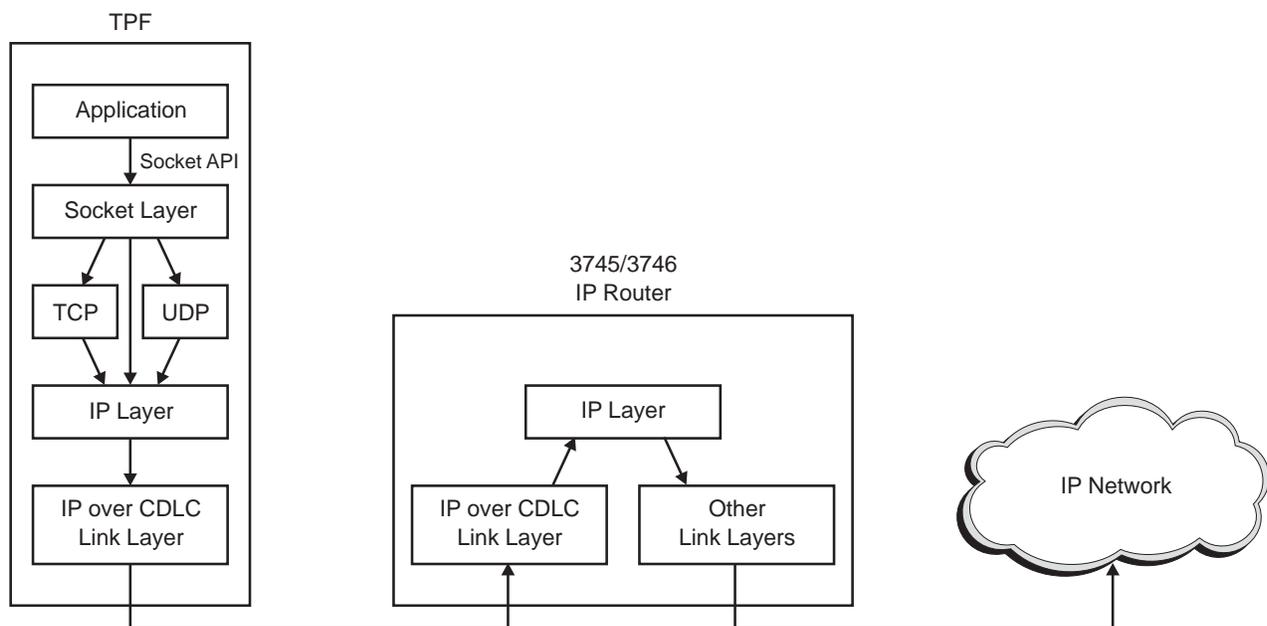


Figure 11. TCP/IP Layers

Figure 12 on page 40 shows the TCP/IP layers that reside in the TPF host and those that reside in the 374x Internet Protocol (IP) routers and the Open Systems Adapter (OSA)-Express card.

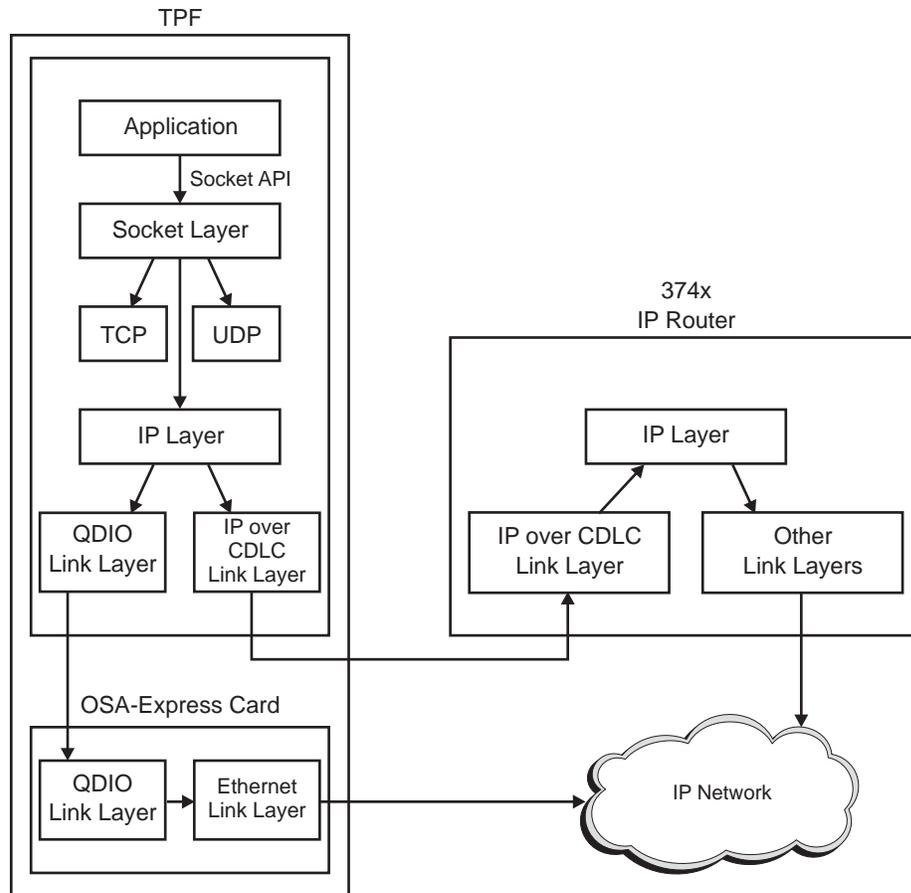


Figure 12. TCP/IP Layers with OSA-Express Support

The following layers reside in the TPF system:

- The socket layer
- The protocol layers, which are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP)
- The IP layer
- Two link layers:
  - IP over channel data link control (CDLC) protocol, which is used to communicate with the IBM 374x routers.
  - The queued direct I/O (QDIO) protocol, which is used to communicate with the Open Systems Adapter (OSA)-Express card.

The following layers reside in the IP router:

- The IP layer
- Multiple link layers.

The IP router performs standard IP routing. The IP routers have no knowledge of sockets. Instead, the routers just forward packets based on the destination IP address of the packet.

---

## Using CDLC IP Routers

One link layer in the TPF system is the IP over channel data link control (CDLC) protocol.

## Configuration Characteristics of CDLC IP Routers

The 3745 and 3746 IP routers have the following configuration characteristics:

- Each connection between the IP router and host processor is through a single subchannel. Data is sent and received through this single subchannel.
- An IP router can have one or more connections (links) to a host processor. If there is more than one connection between a 374x device and a host processor, the TPF host will handle these as connections to different IP routers even though the connections are to the same physical 374x device.
- An IP router can have connections to one or more host processors. This includes connections to multiple host processors in the same loosely coupled complex.
- You can have as many as 200 IP router connections on each TPF host processor.
- A TPF host processor can send both IP traffic and Systems Network Architecture (SNA) traffic to the same 374x device. When this occurs, different subchannels are needed. Across a given subchannel, only one type of traffic can flow (IP or SNA, but not both).

## Data Flow between the CDLC IP Router and the TPF System

The link layer protocol that is used to communicate between the IP router and host processor is IP over CDLC, which is very similar to the SNA over CDLC protocol that TPF host processors use to connect to SNA networks.

The exchange identification (XID) process that is used to activate the connection between the IP router and the TPF system is the same as activating a SNA PU 2.1 link connection. Although the process is the same, the information exchanged is different. Instead of SNA information (for example, control point (CP) name and transmission group (TG) number), IP information (for example, IP addresses) is exchanged.

After the connection is activated, data is transferred using separate read and write channel programs. Multiple packets can be sent or received in a single channel program.

I/O interrupts from IP routers are enabled on all I-streams in the TPF host processor. This allows the interrupt to be processed faster and allows you to take full advantage of TPF tightly coupled support.

## Sample TCP/IP Networks

Figure 13 on page 42 shows a uniprocessor TPF system connected to three IP routers. The IP routers are then connected to the remainder of the IP network.

In this example, there is only one IP address needed for the TPF host (9.123.145.111). Every remote client that wants to connect to the TPF system specifies that IP address on its connect request. A Domain Name System (DNS) *round-robin* of IP addresses or other load balancing methods are not needed for this. The same is true for loosely coupled TPF systems that have only one TPF host processor connected to the IP network.

If the connection to an IP router fails, no sockets are lost because other paths still exist from the TPF host to the remote clients. IP routers are not single points of failures in this example.

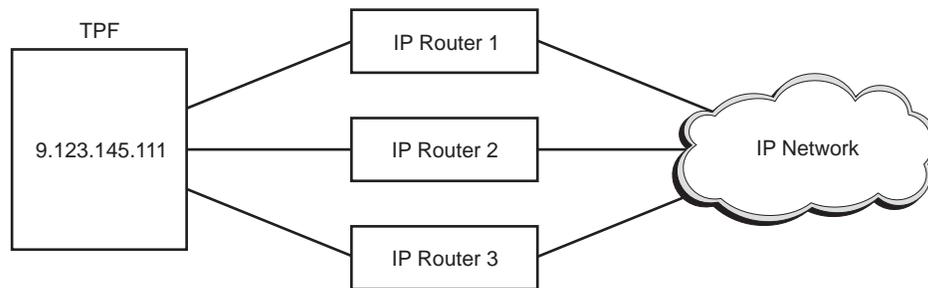


Figure 13. One TPF Host Connected to One IP Network

Figure 14 on page 43 shows a three-way loosely coupled TPF complex connected to two separate IP networks: one being a private intranet and the other the Internet. IP routers 1 and 2 are connected to the intranet. IP routers 3 and 4 are connected to the Internet.

Each TPF host processor must have a unique IP address for each IP network to which it connects. For example, host TPFA uses IP address 25.111.222.82 when connecting to the intranet and IP address 9.123.145.201 when connecting to the Internet. The IP addresses of hosts TPFA, TPFB, and TPFC **must** all be unique.

When a remote client wants to connect to the TPF system, a DNS round-robin of IP addresses or other load balancing method is needed. For example, if a remote client in the Internet wants to connect to the TPF system and does not care which TPF host is selected, you can select IP address 9.123.145.201 (TPFA), 9.123.145.205 (TPFB), or 9.123.145.210 (TPFC). The decision of which IP address to use is done at the client end before the first packet flows into the TPF system.

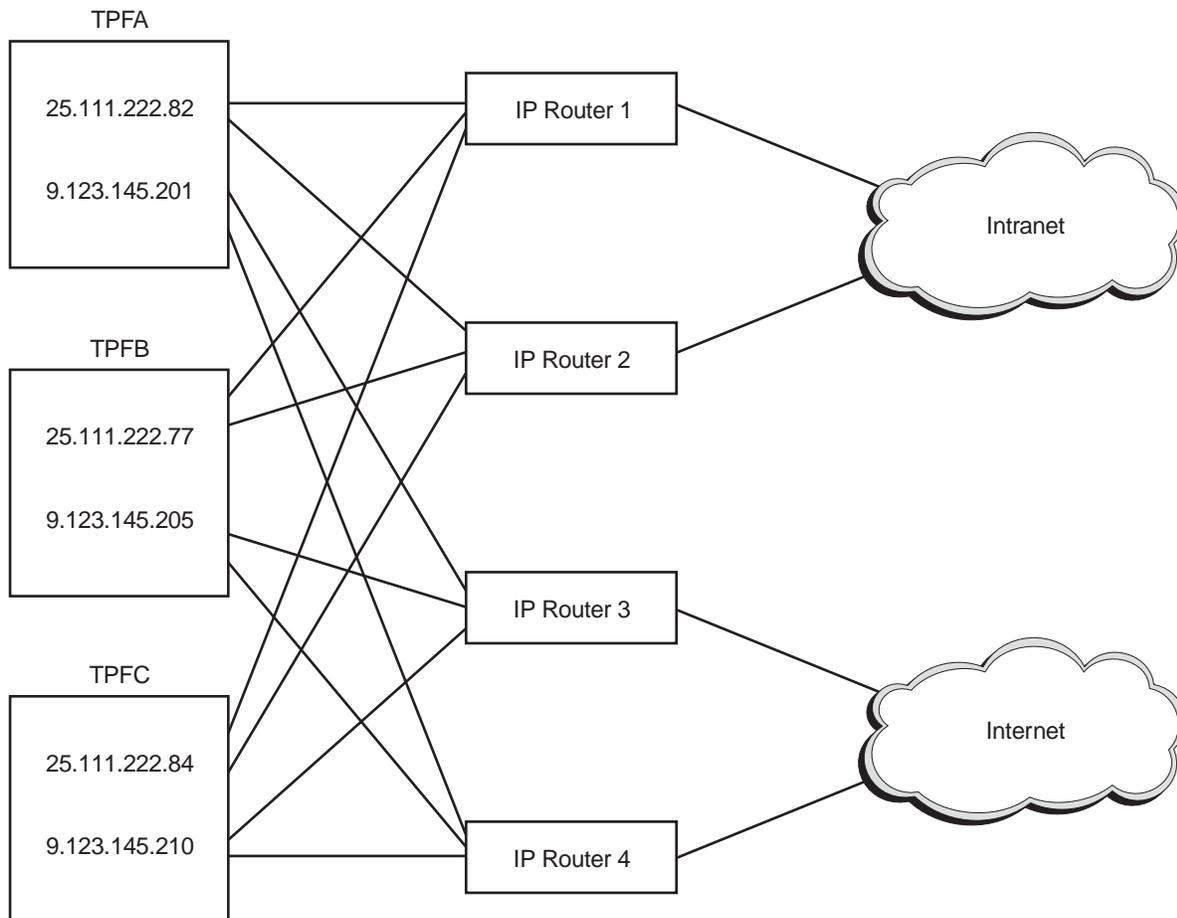


Figure 14. Three TPF Hosts Connected to Two IP Networks

---

## Using OSA-Express Support

Another link layer in the TPF system is the queued direct I/O (QDIO) link layer, which enables communication with the OSA-Express card.

### Configuration Characteristics of the OSA-Express Card

The OSA-Express card has the following configuration characteristics:

- OSA-Express cards are supported by IBM Generation 5 (G5) or later servers. The G5 or G6 server can support a maximum of 12 OSA-Express cards. The zSeries 900 (z900) server can support a maximum of 24 OSA-Express cards. The host (for example, TPF or OS/390) can connect to multiple OSA-Express cards. When the server is logically partitioned (in LPAR mode), all the host LPARs in that server can share the same OSA-Express card.
- Each OSA-Express card supports 240 symbolic device addresses (SDAs). Each connection between the TPF system and OSA-Express requires three SDAs: a write control path, a read control path, and a data path. Each connection between OS/390 and OSA-Express requires at least three SDAs; however, more are allowed because you can define multiple data paths. The unit addresses of the SDAs must be defined in a specific manner; see “Defining OSA-Express Cards to the Processor” on page 66.

- Each OSA-Express card supports a total of 512 IP addresses for all the hosts that share that card.

**Note:** If the total of 512 IP addresses is exceeded, results cannot be predicted.

- You can configure the OSA-Express card by entering the ZOSAE command. See *TPF Operations* for more information on the ZOSAE command.

**Note:** You do not need the OSA/SF feature to configure or use the OSA-Express card.

- Each OSA-Express card has a single port that connects to a Gigabit Ethernet (GbE or GENET) or Fast Ethernet (FENET).

## Data Flow between the OSA-Express Card and the TPF System

Queued direct I/O (QDIO) is the link layer protocol that is used to communicate between the OSA-Express card and host processor (the TPF system). QDIO enables the OSA-Express card and the TPF system to share memory; therefore data transfer does not require channel programs or use the I/O subsystem. IP packets are sent or received on any I-stream.

---

## Components of TCP/IP Native Stack Support

Figure 15 on page 45 provides an overview of the TCP/IP native stack support components.

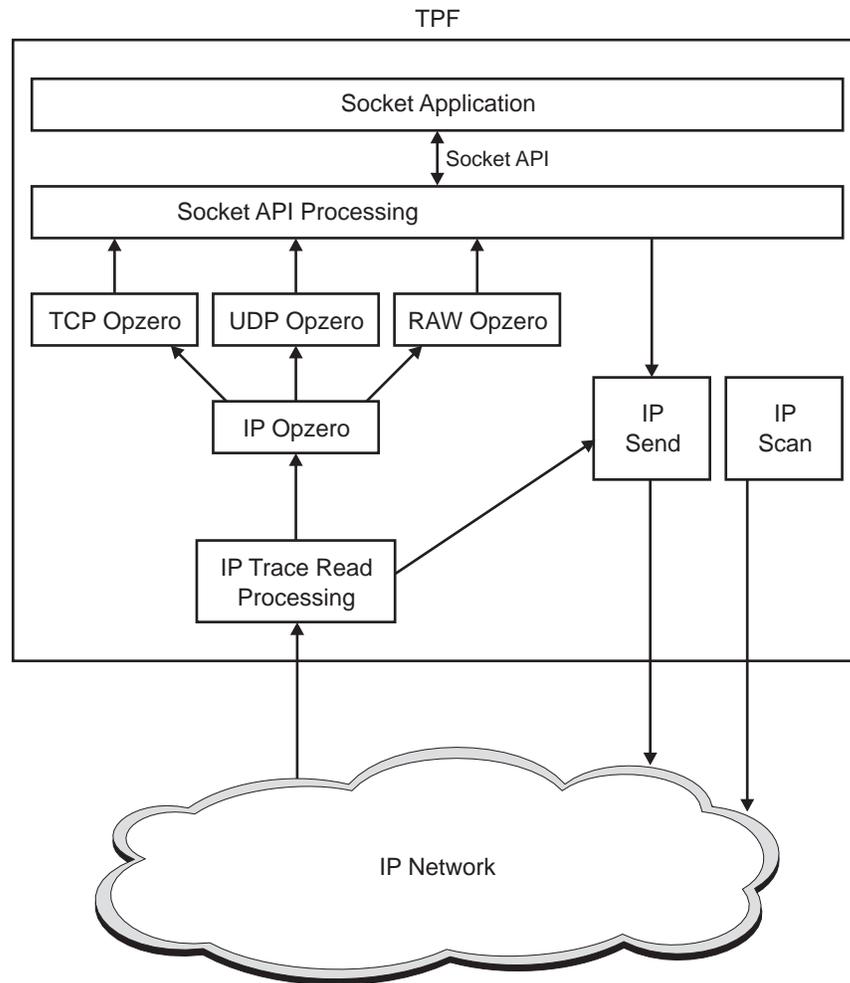


Figure 15. TCP/IP Native Stack Support Components

### Socket application

The user application or TPF middleware that issues socket application programming interface (API) functions.

### Socket API processing

Processes socket API functions. For calls that require packets to be sent out, the IP send component is called. For calls to read data, and the necessary data has already been received from the network, the call is processed completely by this layer.

### IP send

Builds all packets and adds them to the IP output queue to send them to the network, or sends the packets directly to the OSA-Express card. If a large amount of CDLC output packets exist, a write channel program will be issued if an available IP router is found to write out the packets immediately (rather than wait for polling to write out the packets).

### IP scan

Activated several times per second to poll the network (send and receive packets) and retransmit lost TCP output messages.

### IP trace read processing

Activated when packets are received from the IP network. Initial processing of a packet is done to identify the socket that the packet is destined for, process

TCP connection requests, and stand-alone acknowledgments of TCP data. Packets containing user data are passed via the TPF input list to IP Opzero for further processing.

**IP Opzero**

Reassembles fragmented packets and then passes the entire message to the appropriate protocol layer.

**TCP Opzero**

Processes data received for stream sockets. This includes processing out-of-order data and acknowledging the receipt of data.

**UDP Opzero**

Processes data received for datagram sockets.

**RAW Opzero**

Processes data received for RAW sockets. This includes processing ICMP requests (like PING requests), ICMP error messages, and user-created RAW sockets.

---

## Policy Agent

The policy agent is a component in the TCP/IP stack that enforces policy decisions regarding network resources, including access, connection load balancing, and message prioritization. The TPF policy agent provides the following functions:

- Connection load balancing by using the TPF Domain Name System (DNS) server. See “Domain Name System Support” on page 101 for more information.
- Packet filtering to verify that a particular remote client attempting to access a TPF application is allowed to access that application. See “Internet Security” on page 105 for more information about TCP/IP packet filtering firewall support for the TPF system.
- Quality of service (QoS) support for differentiated services. See “TCP/IP Network Services Database Support” on page 111 for more information about support for differentiated services in the TPF system.

---

## Outbound Message Flow

Outbound message flow involves the following steps:

1. The socket application issues a send-type API function call.
2. One or more IP packets are built in the send buffer of the socket.
3. The IP packets are added to the IP output queue, or sent directly to the OSA-Express card.
4. The IP packets are sent to the IP routers connected to the TPF system.

When a socket application issues an API call to send user data, the socket API processing layer verifies to see if the send call can be processed at this time. There is a send buffer associated with each socket, and the size of the send buffer is controlled by the application. If there is enough available space in the send buffer of the socket to build the packets containing the user data, the packets are built and control is immediately passed back to the application with a good return code on the API call.

If there is not enough available space in the send buffer of the socket, the action to take is based on the mode in which the socket is running. If the socket is running in nonblocking mode, control is immediately passed back to the application with a return code indicating that the send operation could not be completed at this time. If

the socket is running in blocking mode, the application entry control block (ECB) is suspended until enough space becomes available in the send buffer of the socket, or until the send times out (based on the send timeout value of the socket).

The type of socket determines when space becomes available in the send buffer for a socket:

- For UDP and RAW sockets, output messages are removed from the send buffer of the socket once the messages have been sent to the network.
- For TCP sockets, output messages remain in the send buffer of the socket until the remote end acknowledges receipt of the data.

Once an output message (IP packet) has been built, control is returned to the application that issued the socket API call. When the packets are actually sent to the network is an independent process. The socket type determines whether or not a packet can be sent to the network right away:

- For UDP and RAW sockets, there is no end-to-end flow control; therefore, output messages are always added to the IP output queue or sent directly to the OSA-Express card when they are built.
- For TCP sockets, the remote end controls the rate at which packets are sent. Each time an acknowledgement is received, it includes a window size indicating how much data on this socket can be sent at this time. When a TCP output message is built, it is added to the IP output queue immediately or sent directly to the OSA-Express card only if the amount of data in the packet is less than or equal to the window size advertised by the remote end.

The TPF system supports differentiated services for outbound messages on an application basis. See “TCP/IP Network Services Database Support” on page 111 for more information.

## Outbound Message Flow for CDLC

Packets are added to the IP output queue when they are ready to be sent to the network and, for TCP sockets, when the remote end is willing to accept the packets. The actual sending of the packets to the network is an independent process that may or may not be triggered by adding a packet to the IP output queue. The I/O write initiate routine is called for the following:

- A packet is added to the IP output queue and the size of the IP output is now large, which means that there are enough packets to fill up a write channel program.
- An I/O operation (read or write) is completed successfully and the size of the IP output queue is large, which means that there are enough packets to fill up a write channel program
- An I/O operation (read or write) is completed successfully, there are messages on the IP output queue but the size of the IP output queue is not large, and the IP router has not indicated that it has many messages to send to the TPF system.
- By the IP scan routine, several times per second, to send output messages so the size of IP output queue never becomes too large.

For a write operation to start, an active IP router must exist that is not in slowdown mode and does not have any I/O operation already in progress. Packets are removed from the IP output queue and sent to the IP router. When the write operation is completed successfully, the packets are copied to the IP trace table if the IP trace function is active for those sockets.

## Outbound Message Flow for OSA-Express

If the OSA-Express connection is active, packets are sent directly to the OSA-Express card by using the QDIO link layer and are not placed on the IP output queue unless the destination is in a remote network and there is no active gateway to that network. In this case, packets are placed on the IP output queue.

If the OSA-Express connection is not active or if all gateways to the remote network are not active, packets are placed on the IP output queue until the connection and gateway become active.

See Defining Gateways on page 58 for more information about gateways.

---

## Inbound Message Flow

Inbound message flow involves the following steps:

1. The CDLC IP router or the OSA-Express card sends packets to the TPF system.
2. The IP trace read processing routine starts to process the packets and, sometimes, processes packets completely.
3. IP Opzero continues processing packets that contain user data.
4. The protocol layer (TCP Opzero, UDP Opzero, or RAW Opzero) either queues the input message off the socket or passes the data to the application if a read-type API call is pending.
5. If a read-type API call is not pending when the data is processed by Opzero, the socket API processing layer dequeues the data and passes it to the application later on when the application does issue a read-type API.

When packets are received from a CDLC IP router or OSA-Express card, each packet is passed to the IP trace read processing routine. This routine determines which socket the packet is for and completely processes certain control information, such as TCP connection requests and standalone acknowledgements of TCP data. Packets are added to the IP trace table if the IP trace function is active for those sockets.

Packets that are not completely processed by the IP trace read processing routine (which includes all packets that contain user data), are passed via the TPF input list to IP Opzero. The main function of IP Opzero is to put fragmented messages back together. When a packet is too large to be sent across a physical network, the packet is split into smaller packets. The final destination must then put the pieces back together before delivering the message to the protocol layer. Whenever possible, tune your network to avoid IP fragmentation to prevent the overhead involved in the IP router splitting up the packet and the IP Opzero overhead involved in reassembling the message.

The protocol layer (TCP Opzero, UDP Opzero, or RAW Opzero) is responsible for queuing the input message or passing it directly to the application if a read-type API call is pending on the socket.

TCP Opzero does the following:

1. If the data is old (which means this is a duplicate packet), the data is discarded.
2. If the data is received out of order (which means the sequence number of the start of this data is not the next expected sequence number), the data is queued until it becomes the next expected piece of data.

3. If there is a `read`, `recv`, or `recvfrom` API call pending for this socket, the suspended ECB is posted. When the application ECB is reactivated, the socket API processing layer copies the data from the link layer read buffer to the application work area specified on the read-type API call. If more data is in the packet than will be received by the application, the excess data is queued in the receive buffer of the socket.
4. If there is an `activate_on_receipt` API call pending for the socket, a new ECB is created, the data is copied to the new ECB, and the application program specified on the `activate_on_receipt` API call is activated. If more data is in the packet than will be received by the application, the excess data is queued in the receive buffer of the socket.
5. If there is no `read`, `recv`, `recvfrom`, or `activate_on_receipt` API call pending for the socket, the data is queued in the receive buffer of the socket.

UDP Opzero does the following:

1. If there is no available space in the receive buffer of the socket to queue the input message, the input message is discarded. Remember, there is no flow control for UDP sockets.
2. If there is a `read`, `recv`, or `recvfrom` API call pending for this socket, the suspended ECB is posted. When the application ECB is reactivated, the socket API processing layer copies the input message from the link layer read buffer to the application work area specified on the read-type API call.
3. If there is an `activate_on_receipt` API call pending for the socket, a new ECB is created, the input message is copied to the new ECB, and the application program specified on the `activate_on_receipt` API call is activated.
4. If there is no `read`, `recv`, `recvfrom`, or `activate_on_receipt` API call pending for the socket, the input message is queued in the receive buffer of the socket.

RAW Opzero does the following:

1. Processes control messages; for example, ICMP ECHO requests (PING messages) and ICMP errors.
2. Copies the input message to the receive buffer of each RAW socket whose protocol matches the protocol of the input message.
3. After copying the input message, if there is a `read`, `recv`, or `recvfrom` API call pending for this socket, the suspended ECB is posted.
4. If there is an `activate_on_receipt` API call pending for the socket, a new ECB is created, the input message is copied to the new ECB, and the application program specified on the `activate_on_receipt` API call is activated.



---

## TPF Control Block Structures

This chapter discusses the following:

- The control block structures used by TCP/IP native stack support
- How to define TCP/IP native stack support to the TPF system
- Storage considerations for TCP/IP native stack support
- Performance and tuning of TCP/IP native stack support
- TCP/IP network configurations.

---

### Socket Block Table Structure

The socket block table consists of four parts:

1. The socket block table header, which contains pointers and statistical information about resources used by TCP/IP native stack support
2. Hash buckets for the IP hash table, which allow a socket block table entry to be easily found, given its IP addresses and port numbers as input
3. Hash buckets for the file descriptor hash table, which allow a socket block entry to be easily found, given its file descriptor as input
4. Socket block table entries, which contain one entry for each active socket that is using TCP/IP native stack support.

The socket block table resides in main storage and is the table accessed and updated most frequently by the TCP/IP native stack support code. All of the information about an active socket is maintained in its socket block table entry.

When the TPF system is brought up, all socket block table entries are in the available pool of resources. Socket block table entries are assigned dynamically as sockets are created. When a socket closes, its socket block table entry is returned to the pool of available resources so that it can be reused.

### Defining the Socket Block Table

The MAXSOCK parameter on the SNAKEY macro in keypoint 2 (CTK2) defines the number of socket block table entries. One socket block table entry is needed for each active socket that uses TCP/IP native stack support.

See *TPF ACF/SNA Network Generation* for more information about the SNAKEY macro.

---

### CDLC IP Configuration Record

The channel data link control (CDLC) IP configuration record contains the following tables:

- The CDLC IP address table, which contains the definitions for each local CDLC IP address associated with this TPF processor
- The CDLC IP device table, which contains the definitions for, and status of, each IP router defined to this TPF processor.

The CDLC IP configuration record resides in main storage and on file. Whenever the record is updated in main storage, the information is also saved on file.

## Defining the CDLC IP Network Configuration

Use the ZTTCP commands to define, display, change, or delete CDLC IP network configuration information.

See “Local IP Addresses” on page 60 and “Defining CDLC IP Connections” on page 65 for more information about defining the CDLC IP network configuration to the TPF system.

See *TPF Operations* for more information about the ZTTCP commands.

---

## CDLC IP CCW Area Table

The CDLC IP channel command word (CCW) area table contains one entry for each active IP router that uses CDLC protocol. Channel programs, device status, and read buffer information reside in this table.

## Defining CDLC IP CCW Area Table Resources

The MAXIPCCW parameter on the SNAKEY macro in CTK2 defines the number of IP CCW area table entries. The size of each IP CCW area is 4096 bytes.

Each active IP router has two sets of read buffers assigned to it for reading packets from the network. The IPRBUFFS parameter on the SNAKEY macro in CTK2 defines the number of read buffers in each set. The IPRBUFSZ parameter on the SNAKEY macro in CTK2 defines the size of read buffer.

---

## OSA Configuration Record

The OSA configuration record (OCR) consists of the following tables:

- The OSA definition table, which contains the definitions and status of the OSA-Express connections on this processor.
- The OSA IP address table, which contains the definitions and status of the IP addresses associated with the OSA-Express connections on this processor.

The OSA configuration record resides in main storage and on file. Whenever the record is updated in main storage, the information is also saved on file.

---

## OSA Control Block Table

The OSA control block table resides in main storage and is the table accessed and updated most frequently by the OSA-Express support code. All the information about an active OSA-Express connection is maintained in its control block table entry.

The MAXOSA parameter on the SNAKEY macro in keypoint 2 (CTK2) defines the maximum number of OSA-Express connections that can be active on the TPF system. One OSA control block table entry is needed for each active OSA-Express connection on this processor.

---

## OSA Shared IP Address Table

The OSA shared IP address table (OSIT) resides only on file and contains the status of all OSA IP addresses in the complex. To define the #OSIT fixed file records, see *TPF System Generation*.

---

## OSA Read Buffers

The OSA read buffers are used to transfer data to the TPF system from an OSA-Express connection. Each OSA-Express connection has its own set of read buffers allocated in main storage.

### Defining OSA Read Buffers

The OSABUFF parameter on the SNAKEY macro in CTK2 defines the number of 64-KB read buffers assigned for each OSA-Express connection. You can specify a value of 16, 32, or 64 for the OSABUFF parameter. See *TPF ACF/SNA Network Generation* for more information about the SNAKEY macro.

The amount of storage required for each valid OSABUFF value is as follows:

Number of OSA Read Buffers (OSABUFF)	Storage Required
16	1 MB × MAXOSA parameter value
32	2 MB × MAXOSA parameter value
64	4 MB × MAXOSA parameter value

Consider the following when determining how many OSA read buffers to define:

- The number of messages received per second
- The average size of the messages received.

Increasing the OSABUFF parameter is only necessary for systems with significant message rates. In a low-volume TPF system, the default of 16 buffers should be enough. However, in a high-volume production system (for example, a system that receives more than 2000 messages per second), it is best to increase the number of buffers.

The message rate at which you decide to increase the number of buffers can vary based on processor speed, number of I-streams, and the environment in which the TPF system is running. One way to determine whether you need to increase the number of buffers is to look at the number of messages that are received out of order. If the percentage of messages received out of order increases as the message rate increases, it can be an indication that the buffers currently available to the OSA are being filled and the value of the OSABUFF parameter should be increased.

---

## IP Message Table

The IP message table (IPMT) contains the following:

- IP packets built by the TPF system that have not yet been sent to the network
- IP packets for TCP sockets that have been sent to the network and are waiting for an acknowledgment from the remote end
- IP packets received from the network that have not been delivered to the socket application yet
- TCP data received out of order from the network
- IP fragments received from the network.

The IPMT resides in main storage and consists of a pool of 4-KB entries that are assigned dynamically on demand.

Output messages reside in the send buffer of a socket. Input messages reside in the receive buffer of a socket. The IPMT is the physical storage used by send and receive buffers. IPMT entries are not preallocated to send and receive buffers. Instead, IPMT entries are assigned only when needed. For example, assume the receive buffer size for a socket is 100 KB and there are no input messages queued for this socket. For this, there are no IPMT entries being used for input messages. Next, assume a 500-byte message arrives for this socket. An IPMT entry will be assigned to the receive buffer of the socket and the 500-byte message will be copied into that entry. When the application reads the 500-byte message, the IPMT entry will be returned to the system.

## Defining the IP Message Table

The IPMTSIZE parameter on the SNAKEY macro in CTK2 defines the number of 4-KB entries in the IPMT. The IPMT is usually the TCP/IP native stack support table that requires the most main storage. The following factors must be taken into consideration when determining the size of the IPMT:

- Message rate (number of packets sent and received per second)
- Average size of a packet
- Round-trip time for TCP sockets, which means how long it takes, on average, to receive responses from remote partners.

---

## IP Routing Table

The IP routing table (IPRT) contains the following:

- The IP routing table header, which provides pointers and statistical information about the IP routing table and its core and fixed file entries.
- Unique IP routing table entries, each associating a remote IP address or subnet with a local TPF IP address or a next hop gateway. The size of each IP routing table entry is 64 bytes.

The IPRT resides in main storage and is filed out to DASD whenever entries are added, deleted, or modified. During an initial program load (IPL), the IPRT is rebuilt from the #IPRTE fixed file records. See *TPF ACF/SNA Network Generation* for more information about the #IPRTE 4-KB fixed file records.

## Defining the IP Routing Table

The MAXRTE parameter of the SNAKEY macro in keypoint record 2 (CTK2) defines the number of IP routing table entries. You can change the MAXRTE parameter by entering the ZNKEY command; the new values will be used when the TPF system is IPLed. The ZTRTE command manages the IP routing table entries, giving you the ability to add, delete, modify, and display them.

See *TPF Operations* for more information about the ZNKEY and ZTRTE commands. See *TPF ACF/SNA Network Generation* for more information about the SNAKEY macro.

---

## Tuning TCP/IP Native Stack Support

This section describes how to tune parameters in the TPF system and in the network.

## Tuning Major Control Block Structures

The ZTTCP DISPLAY STATS command display includes the maximum number of socket block entries and maximum number of IPMT entries that were in use at any time since the last IPL of the TPF system. Monitor this information to see if either control block is approaching the limit that you defined and, if so, increase the number of control block entries:

- To increase the number of socket block entries, increase the value of the MAXSOCK parameter on the SNAKEY macro in CTK2.
- To increase the number of IPMT entries, increase the value of the IPMTSIZE parameter on the SNAKEY macro in CTK2.

## Tuning the IP over CDLC Link Layer

Examine IP trace data during a peak traffic period to see how many IP packets are sent and received in each write and read channel program. The value of the IPRBUFFS parameter on the SNAKEY macro in CTK2 defines the maximum number of packets that can be sent or received in one channel program. Packets that were sent or received in the same channel program will have the same time stamp in the IP trace display. If the average number of packets in each write or read channel program is at or near the maximum that you defined, increase the value of the IPRBUFFS parameter so more data can be sent in each channel program.

The value of the IPRBUFSZ parameter on the SNAKEY macro in CTK2 defines the maximum size of a packet that can be sent or received. For stream sockets, the maximum amount of user data that can be sent or received in a packet is the value of IPRBUFSZ minus the combined size of the link header, IP header, and TCP protocol header. Code IPRBUFSZ based on the average size of TCP messages (amount of user data in each message) in your network as follows:

Average Message Size	IPRBUFSZ Value
1–976	1024
977–2000 bytes	2048
Over 2000 bytes	4096

Even if you set IPRBUFSZ to a value that is smaller than your average message size, you can still send large messages. However, those messages will be sent in smaller pieces causing more packets to flow.

Setting IPRBUFSZ to a large value does not guarantee that large packets will flow. When a stream socket is started, the two sides negotiate the maximum size of a packet that can be sent on the socket. If the remote node suggests a maximum packet size that is less than the value of IPRBUFSZ, the value suggested by the remote node is used.

## Tuning the IP Network

The ZTTCP DISPLAY STATS command display shows the number of IP fragments received. If the display shows that a large amount of fragmented messages are being received, examine the network and take the following actions to avoid IP packets from being fragmented:

1. Identify the intermediate network device that is fragmenting the packets and increase the maximum packet size of that network device to a value large enough to avoid fragmentation.

2. Lower the value of the maximum packet size used by TCP connections with the TPF system by doing the following:
  - For CDLC, enter the ZTTCP CHANGE command, specifying a value for the maximum packet size (MPS) parameter that is small enough to avoid IP packet fragmentation.
  - For OSA, enter the ZOSAE command with the MODIFY parameter specified, specifying a value for the maximum transmission unit (MTU) parameter that is small enough to avoid IP packet fragmentation.

The ZTTCP DISPLAY STATS command display also shows the number of TCP packets that were retransmitted by the TPF system. If the number of retransmitted packets is high, this usually indicates congestion problems in the network that should be investigated.

---

## Performance

There are counters that provide information about the number of messages, bytes, and packets sent and received for each TCP/IP application. This information is provided in the reports that are generated by data collection and reduction. The counts are incremented differently based on whether or not the application is defined in the TCP/IP network services database. See “TCP/IP Network Services Database Support” on page 111 for more information about this data collection and how to define your applications in the TCP/IP network services database. See *TPF System Performance and Measurement Reference* for more information about data collection and reduction in general.

The ZSTAT command display shows the total number of weighted TCP/IP messages. See “TCP/IP Network Services Database Support” on page 111 for more information about weighted TCP/IP messages. See *TPF Operations* for more information about the ZSTAT command.

The ZTTCP DISPLAY STATS command display shows the total number of packets sent and received by the TPF system. You can display this information periodically and calculate the rate at which packets are transferred between the TPF system and IP routers.

The ZVIPA command displays statistical information about your TPF system and can be used to balance the OSA TCP/IP workload. For more information, see “Workload Balancing Using Movable VIPAs” on page 69.

---

## TCP/IP Network Configurations

To connect the TPF system to multiple TCP/IP networks, you can use the IP routing table (IPRT). You can set up IPRT entries to associate local IP addresses with a remote IP address or subnet of addresses.

When you use OSA-Express support, you can set up IPRT entries to associate remote networks and the gateways to them.

### Selecting the Local IP Address

When a TPF TCP client application attempts to connect to a remote server, a bind function for a local TPF IP address may not have been explicitly issued. If this occurs, TCP/IP native stack support uses the IPRT to select a local IP address. This local IP address determines the set of channel-attached IP routers or OSA-Express connections that can be used for the connection to the remote server.

The IPRT entry that contains the best match for the IP address of the remote server is used. The IPRT entry with the most specific network mask becomes the best match for the IP address. The network mask represents a subnet of IP addresses or a specific IP address if it is equal to 255.255.255.255. If multiple entries qualify as the best match, the TPF system selects one entry on a round-robin basis. If a best match cannot be found, the default local IP address is used. If an entry is the best match, but it has a local TPF IP address that does not have active IP routers or OSA-Express connections, the entry is not selected.

### Choosing Network Paths

In the following example, the TPF system is connected to two IP routers. IP router A has a local IP address of 4.4.4.4 and is connected to a remote network having a subnet IP address of 8.8.8.0. IP router B has a local IP address of 5.5.5.5 and is connected to a different remote network that has a subnet IP address of 10.10.10.0. The default local address is 5.5.5.5. There are no entries in the IPRT.

A client application wants to establish a connection to a remote server that has an IP address of 8.8.8.1. The client application issues a connect function without issuing a bind function. The TPF system searches the IP routing table for a remote address of 8.8.8.1 to determine which local IP address to use when sending the packet. The TPF system determines that the IPRT is empty and continues processing using the default local IP address of 5.5.5.5. The packet for the remote server is sent out across IP router B (5.5.5.5.); however, a connection is not established because IP router B is not connected to the 8.8.8.0. network.

You can use the IP routing table to resolve this condition. Enter the ZTRTE command with the ADD parameter specified to add two entries to the IPRT.

```
ZTRTE ADD RIP-8.8.8.1 LIP-4.4.4.4 NETMASK-255.255.255.255
ZTRTE ADD RIP-10.10.10.0 LIP-5.5.5.5 NETMASK-255.255.255.0
```

The IPRT now contains the following information:

Remote IP Address	Network Mask	Local IP Address
8.8.8.1	255.255.255.255	4.4.4.4
10.10.10.0	255.255.255.0	5.5.5.5

The client application again tries to establish a connection to the remote server that has an IP address of 8.8.8.1. The client application issues a connect function without issuing a bind function. The TPF system searches the IP routing table for a remote address of 8.8.8.1 to determine which local IP address to use when sending the packet. The TPF system finds an entry in the IPRT indicating that the local IP address of 4.4.4.4 is associated with IP router A and should be used for this connection. The packet for the remote server is sent out across IP router A (4.4.4.4) and a successful connection is established between the client application (4.4.4.4) and the remote server (8.8.8.1).

If more than one entry exists for a remote IP address or subnet of remote IP addresses, the TPF system will search through the IPRT entries for each connection. It is possible to have more than one entry for the same IP address only if the local IP address is unique from all other entries with the same remote IP address. For example, a third IP router, IP router C, with an IP address of 7.7.7.7 is added to the TPF system. IP router C is also connected to the remote network of 8.8.8.0. Another entry can be added to the IPRT for the remote server (8.8.8.1). Now you have multiple paths in the IP routing table to get to the remote server with

the IP address of 8.8.8.1. The IPRT now looks like this:

Remote IP Address	Network Mask	Local IP Address
8.8.8.1	255.255.255.255	7.7.7.7
8.8.8.1	255.255.255.255	4.4.4.4
10.10.10.0	255.255.255.0	5.5.5.5

The IPRT can be created and managed by the ZTRTE command. See *TPF Operations* for more information about the ZTRTE command.

## Defining Gateways

When the TPF system sends an IP packet to a destination that is on the network where the OSA-Express card is connected, the packet is sent directly to the destination. However, if the destination of the packet is a remote network, the TPF system must inform the OSA-Express card through which gateway to send the packet. For this, a gateway is a router that connects the local network (to which the OSA-Express card is attached) to remote networks. You can define up to two default gateways for each OSA-Express connection to your TPF system by entering the ZOSAE command specifying the DEFINE or MODIFY parameters and the GATEWAY1 or GATEWAY2 parameter.

For complex networks where you want to use specific gateways based on the remote destination, you can define IP routing table entries to specify which gateways the TPF system will use when sending packets to a specific remote node or network. Enter the ZTRTE command with the following parameters to define an IP routing table entry:

- NEXTHOP, specifies the IP address of the gateway on the local network
- RIP, specifies the IP address of the remote node or network
- NETMASK, specifies the subnet mask that is applied to the RIP parameter.

For example, your local network is 1.1.1.x and has five gateways whose IP addresses are 1.1.1.10, 1.1.1.11, 1.1.1.30, 1.1.1.40, and 1.1.1.43. You want all traffic destined for remote network 3.3.3.x to go through gateways 1.1.1.30 and 1.1.1.40. To do this, enter the following:

```
ZTRTE ADD RIP-3.3.3.0 NETMASK-255.255.255.0 NEXTHOP-1.1.1.30  
ZTRTE ADD RIP-3.3.3.0 NETMASK-255.255.255.0 NEXTHOP-1.1.1.40
```

When multiple IP routing table entries exist for the same destination, the TPF system will use a round-robin method for selecting a gateway.

## Choosing a Gateway

When a socket is created with a node that resides in a remote network and the TPF system has the first packet to send on that socket, a gateway is selected as follows:

1. The IP routing table is searched to find an entry that matches the remote destination and whose gateway resides on the same local network to which the OSA-Express card is connected. If a match is found, that gateway is used.
2. If an IP routing table entry is not found, one of the two default gateways for the OSA-Express connection is used if there is an active default gateway.
3. If an active gateway does not exist, the packet is queued until one of the default gateways becomes active or until you add a new default gateway. Once a gateway is selected, that gateway is used for the life of the socket unless the gateway fails; however, that gateway can direct the TPF system to use a different gateway by sending an ICMP redirect message.

---

# Operator Procedures for TCP/IP Native Stack Support

This chapter describes the tasks that an operator can perform for TCP/IP native stack support.

---

## Configuring a TPF System

To use TCP/IP native stack support, you must do the following:

- If you are using channel data link control (CDLC), do the following:
  1. Code the MAXSOCK, MAXIPCCW, IPRBUFFS, IPRBUFSZ, IPMTSIZE, and SOCKSWP parameters of the SNAKEY macro and reload keypoint 2 (CTK2). See “Enabling TCP/IP Native Stack Support” on page 60.
  2. Define the local IP addresses by entering the ZTTCP DEFINE command.
  3. Define the IP routers by entering the ZTTCP DEFINE command.
  4. Activate the IP routers by entering the ZTTCP ACTIVATE command.
- If you are using Open Systems Adapter (OSA)-Express, do the following:
  1. Code the MAXSOCK, IPMTSIZE, SOCKSWP, and MAXOSA parameters of the SNAKEY macro and reload keypoint 2 (CTK2). See “Enabling TCP/IP Native Stack Support” on page 60.
  2. Define and allocate the #OSIT fixed file record. For more information about calculating the number of records to allocate, see *TPF System Generation*.
  3. Define the OSA-Express connections to the TPF system by entering the ZOSAE command with the DEFINE or MODIFY parameter specified.
  4. Define the virtual IP addresses (VIPAs) by entering the ZOSAE command with the ADD parameter specified.
  5. Activate the OSA-Express connections by entering the ZTTCP ACTIVATE command.

To use IP routing table support, you must do the following:

1. Have TCP/IP native stack support defined on your TPF 4.1 system.
2. Code the new MAXRTE parameter of the SNAKEY macro and reload keypoint 2 (CTK2). See “Enabling TCP/IP Native Stack Support” on page 60.
3. Define and allocate the #IPRTE fixed file record. For more information about calculating the number of records to allocate, see *TPF ACF/SNA Network Generation*.
4. Define the IP routing table entries by entering the ZTRTE command with the ADD parameter specified.
5. Activate the IP routers or OSA-Express connections by entering the ZTTCP ACTIVATE command.
6. Activate TCP client socket applications on the TPF 4.1 system to use IP routing table support.

For more information about the ZTTCP and ZTRTE commands, see *TPF Operations*.

To use individual IP trace support, you must do the following:

1. Have TCP/IP native stack support defined on your TPF 4.1 system.
2. Code the IPTRCNUM and IPTRCSIZ parameters of the SNAKEY macro and reload keypoint 2 (CTK2). See “Enabling TCP/IP Native Stack Support” on page 60 for more information.

3. Define the individual IP traces by entering the ZINIP command.

For more information about the SNAKEY macro, see *TPF ACF/SNA Network Generation*. For more information about the ZINIP command, see *TPF Operations*.

---

## Enabling TCP/IP Native Stack Support

To enable TCP/IP native stack support, code the following SNAKEY macro parameters in CTK2:

### **MAXSOCK**

The number of socket block entries. See “Defining the Socket Block Table” on page 51.

### **MAXIPCCW**

The number of IP channel command word (CCW) area entries. See “Defining CDLC IP CCW Area Table Resources” on page 52.

### **IPRBUFFS**

The number of read buffers per IP router. See “Defining CDLC IP CCW Area Table Resources” on page 52 for more information.

### **IPRBUFSZ**

The size of each read buffer for IP routers. See “Defining CDLC IP CCW Area Table Resources” on page 52.

### **IPMTSIZE**

The size of the IP message table (IPMT). See “Defining the IP Message Table” on page 54.

### **IPTRCNUM**

The maximum number of individual IP traces you can define. See “Using the Individual IP Trace Function” on page 76.

### **IPTRCSIZ**

The size of each individual IP trace. See “Using the Individual IP Trace Function” on page 76.

### **SOCKSWP**

The socket sweeper interval. See “Socket Sweeper Support to Close Inactive Sockets” on page 125.

### **MAXRTE**

The maximum number IP routing table (IPRT) entries. See “Defining the IP Routing Table” on page 54.

### **MAXOSA**

The maximum number of OSA-Express connections that can be active on the TPF system. See “OSA Control Block Table” on page 52.

---

## Local IP Addresses

Your network administrator must assign at least one IP address to each TPF host that connects to an IP network. If a TPF host connects to only one IP network, only one IP address is needed for the TPF host. However, you can assign multiple IP addresses to a TPF host even if the TPF host connects to only one IP network.

In a loosely coupled TPF system, each TPF processor must have unique IP addresses so that packets are routed to the correct TPF processor. However, movable VIPAs can be defined on every processor but a given VIPA can only be active on one processor at a time.

If a TPF host connects to more than one physically separate IP network, the TPF host will have a different IP address for its connections to the different networks. For example, if a TPF host connects to three physically separate IP networks, that TPF host will have at least three IP addresses defined.

## Default Local IP Address

The first local IP address that you define to a TPF host becomes the default local IP address for that host. The default local IP address is used when a socket client application in the TPF system sends data and has not informed the TPF system which local IP address to use. For example, a TCP client application running in the TPF system issues a `connect` call without issuing a `bind` call first. When this occurs, the TPF system must select a local IP address to use. The IPRT is searched to find a local IP address to use that is based on the destination. If an entry for the destination does not exist, the default local IP address is used.

If a TPF host has more than one local IP address defined and you have socket client applications running in your TPF system, any client application that needs to use a local IP address other than the default must issue a `bind` call to bind that socket to the desired local IP address or to define entries in the IPRT. Client applications that want to use the default local IP address do not need to issue the `bind` call.

For information about local IP addresses, see [Selecting the Local IP Address](#) on page 56.

To define the default local IP address, specify the `DEFAULT` parameter on the `ZTTCP DEFINE` command or specify the `ZOSAE` command with the `DEFINE`, `MODIFY`, or `ADD` parameter. These define the local IP address to the TPF system. You can change which IP address is the default by entering the `ZTTCP CHANGE` command and specifying the `DEFIP` parameter.

## Maximum Packet Size

When a TCP connection is started, part of the handshake process determines the maximum packet size (MPS) that can be sent on the socket. Typically, when the client sends the connection request, the suggested MPS sent is the smaller of the following values:

- The buffer size in the node where the client resides
- The maximum size packet supported by the network to which the client directly connects.

When the server receives the connection, the MPS value that will be used by the socket is the smallest of the following values:

- The suggested MPS value sent by the client
- The buffer size in the node where the server resides
- The maximum size packet supported by the network to which to server directly connects.

Assume the client and server nodes both support 4-KB buffer sizes and the networks to which they are directly connected support 4-KB packets. However, an intermediate network along this connection supports only 1-KB packets. If the TCP handshake process does not account for the intermediate network, the MPS value for the socket will be 4 KB. Every 4-KB packet sent by the client to the server will be fragmented by the intermediate network into four 1-KB packets and the server node will have to put the pieces back together before delivering the data to the

application. Because this is an expensive process that can impact throughput and network performance, IP fragmentation should be avoided whenever possible.

When you define a local IP address to your TPF system, you can define the upper limit for the MPS value used by TCP connections connected to a local IP address. When you define an OSA-Express connection, you can define the upper limit for the maximum transmission unit (MTU) value used by all IP addresses defined to that OSA-Express connection. In the previous example, because the intermediate network supports only 1-KB packets, enter the ZOSAE command with the DEFINE parameter specified and a value of 1024 specified for the MTU parameter; or enter the ZTTCP DEFINE command, which defines the local IP address. You can change the MTU parameter value by entering the ZOSAE command with the MODIFY parameter specified, or you can enter the ZTTCP CHANGE command and specify the MPS parameter. This change does not affect existing sockets; it only applies to subsequently created sockets.

---

## Types of TPF Local IP Addresses

The three types of local IP addresses are:

- Channel data link control (CDLC)
- Real OSA IP address
- Virtual IP address (VIPA).

A maximum of 250 IP addresses can be defined for all OSA-Express connections in the TPF system. This includes real OSA IP addresses and VIPAs. You can define as many as 16 local CDLC IP addresses for each TPF host. A TPF IP address is associated with either 374x IP routers or with an OSA-Express card; therefore, a socket which is bound to a specific local IP address uses either 374x connections or OSA-Express connections, but not both.

## CDLC Addresses

CDLC local IP addresses are associated with 374x IP routers.

### Defining CDLC IP Addresses

Enter the ZTTCP DEFINE command to define the IP addresses of the TPF host. You must do this before you define the IP routers to the TPF system.

### Deleting CDLC Local IP Addresses

If your network configuration changes and you need to delete a local IP address on your TPF host, enter the ZTTCP DELETE command. See *TPF Operations* for more information about the ZTTCP DELETE command.

To delete the local IP address, no IP router can be associated with this local IP address. If there are IP routers associated with this local IP address, you must first change or delete the IP router definitions before you delete the local IP address.

The default local IP address cannot be deleted if one or more local IP address is defined. If more than one local IP address (including CDLC, real OSA, and VIPA) is defined and you want to delete the IP address that is currently the default, do the following:

1. Change which local IP address is the default by entering the ZTTCP CHANGE command specifying the DEFIP parameter.
2. Delete the desired local IP address by entering the ZTTCP DELETE command.

## Restricting CDLC IP Addresses

When a TPF host connects to more than one physically separate IP network, the TPF host has more than one local IP address defined. You can define a CDLC local IP address to be restricted to prevent remote clients in one IP network from accessing TPF applications that reside in a different IP network.

For example, assume a TPF host is connected to a private intranet and to the Internet. The TPF system uses IP address 1.1.1.1 to connect to the intranet and IP address 2.2.2.2 to connect to the Internet. IP address 1.1.1.1, port 5001, represents a server application in your TPF system that can be accessed only by clients in the intranet. To prevent clients in the Internet from accessing this application, define IP address 1.1.1.1 as restricted in your TPF system. For example, if an IP packet is received by the TPF system with a destination in the packet of IP address 1.1.1.1 and the packet is not received by an IP router connected to IP address 1.1.1.1, the TPF system will then reject the IP packet.

To define an IP address as restricted, specify `RESTRICT=YES` on the `ZTTCP DEFINE` command. You can change an IP address to be restricted by specifying `RESTRICT=YES` on the `ZTTCP CHANGE` command.

See *TPF Operations* for more information about the `ZTTCP` commands.

## Real OSA IP Addresses

Real OSA IP addresses are associated with OSA-Express connections.

### Defining Real OSA IP Addresses

You must define one unique real OSA IP address of the TPF system for each OSA-Express connection and this IP address must be in the subnet of the Ethernet connected to the OSA-Express card. For example, if the Ethernet is network 9.117.249.x (with a subnet mask of 255.255.255.0), the real OSA IP address of the TPF host could be 9.117.249.31 but could not be 9.117.222.31. The real OSA IP address of the TPF host is defined to the TPF system by entering the `ZOSAE` command with the `DEFINE` or `MODIFY` parameter specified.

Real OSA IP addresses are tied to a fixed connection and cannot be moved. TPF applications can bind to a local IP address that is either a real OSA IP address or a VIPA. A real OSA IP address for an OSA-Express connection or a VIPA can also be defined as the default local IP address of the TPF system. If sockets are bound to a real OSA IP address and the OSA-Express connection fails, the sockets will fail. Therefore, it is recommended that you use VIPAs in your production system.

### Deleting Real OSA IP Addresses

You can delete the real OSA IP address of the TPF host and all of the VIPAs associated with the connection by entering the `ZOSAE` command with the `DELETE` parameter specified. When the `ZOSAE` command is entered, the OSA-Express connection must be inactive, and must not have a backup defined. If the real OSA IP address or any VIPA associated with the OSA-Express connection is defined as the default IP address, and there are other local IP addresses (CDLC or OSA) defined, you cannot delete it. See *TPF Operations* for more information about the `ZOSAE` command.

## Virtual IP Addresses (VIPAs)

In addition to CDLC and real OSA IP addresses, VIPAs can be associated with the TPF system across an OSA-Express connection. VIPAs are not fixed and can be

moved from one OSA-Express connection to another or from one physical processor to another in the same loosely coupled complex.

If an OSA-Express card fails, or the switch or router connected to the card fails, any TPF VIPAs assigned to that card automatically swing to the alternate OSA-Express connection on the same processor if one is defined and is active. This enables sockets to remain active and eliminates single points of failure in network-attached hardware. The VIPAs cannot reside in the same subnet as the real IP address of the OSA-Express connection or the real IP address of the alternate OSA-Express connection if one is defined.

## Types of VIPAs

**Static VIPAs:** A static VIPA always resides on one specific TPF processor in the loosely coupled complex. A static VIPA can swing from one OSA-Express connection to another, but always on the same processor. Use static VIPAs to access processor unique TPF applications.

**Movable VIPAs:** A movable VIPA can be defined on more than one processor in a loosely coupled complex, but is active on only one TPF processor at a time. You must define the VIPA as movable to all the processors that may potentially use it. Use movable VIPAs to access processor shared applications and to load balance TCP/IP traffic in the complex.

If you move a VIPA from one processor to another, all existing sockets using that VIPA will fail; when the remote clients reconnect to that VIPA, new sockets will be established on the new processor where that VIPA is now active. By moving a VIPA, you move all traffic for the remote users connected to that VIPA from one processor to another. A VIPA can be moved in the following ways:

- Automatically, when a TPF processor fails if directed to do so by the UVIP user exit
- Manually, through one of the following methods:
  - By an operator entering the ZVIPA command
  - By an application program using the VIPAC macro
  - By an application program using the `tpf_vipac` C function.

**Note:** You cannot use movable VIPAs to communicate between processors in the same loosely coupled complex if both processors have the same movable VIPA defined.

## Defining VIPAs to an OSA-Express Connection

To define a VIPA to an OSA-Express connection, enter the ZOSAE command with the ADD parameter specified and then specify the type of VIPA you want by doing the following:

- To define a static VIPA, specify the STATIC parameter.
- To define a movable VIPA, **do not** specify the STATIC parameter.
- Use the DEFAULT parameter to indicate that the VIPA is the default local IP address.

When you define a VIPA, it is defined to a pair of OSA-Express connections (primary and alternate, if the alternate is defined), but it is only active on one OSA-Express connection at a time.

VIPAs can be defined to an active OSA-Express connection and the VIPAs can be used immediately.

## Displaying VIPAs

To display a VIPA, do one of the following:

- Enter the ZVIPA command with the DISPLAY parameter specified to display one or more VIPAs and to display which TPF processor currently owns the VIPAs.
- Enter the ZVIPA command with the IP parameter specified to display on which CPUs the VIPA is currently defined.

See *TPF Operations* for more information about the ZVIPA command.

## Deleting VIPAs

To delete a VIPA definition on a processor, enter the ZOSAE command with the REMOVE parameter specified. Once you do this, that VIPA is disassociated from its OSA-Express connection and is deleted from the processor on which the message was issued. Enter the ZOSAE command with the REMOVE parameter specified only when the OSA-Express connection associated with the VIPA and its backup OSA-Express connection are not active.

You cannot delete a VIPA when it is defined as the default local IP address.

See *TPF Operations* for more information about the ZOSAE and ZVIPA commands.

---

## CDLC IP Connections

After you have defined the local IP addresses for your TPF system, you must define the IP routers.

### Defining CDLC IP Connections

You can define the IP routers by entering the ZTTCP DEFINE command. For each IP router, specify its symbolic device address (SDA) and the local (TPF) IP address to which it connects. The IP address of the IP router itself is not defined to the TPF system. Instead, this information is found dynamically when the connection to the IP router is activated.

Refer to an IP router by its SDA. The SDA is the unique token that identifies an IP router and is used as input to the ZTTCP commands, which define, change, display, delete, activate, trace, or deactivate an IP router.

Multiple IP routers can be connected to the same local IP address in a TPF host. This enables a TPF host to present a single interface (one IP address) to the remainder of the network.

You can change the TPF IP address associated with the IP router by entering the ZTTCP CHANGE command. However, the IP router must be deactivated before making this change. You can define as many as 200 IP routers per TPF host.

### Activating and Deactivating CDLC IP Routers

IP routers are activated by entering the ZTTCP ACTIVATE command and they are deactivated by entering the ZTTCP INACTIVATE command. You can activate or deactivate all IP routers, one specific IP router, or all IP routers associated with a specific TPF IP address.

When you enter a ZTTCP ACTIVATE command, the IP router is marked as active in the IP configuration record. This causes the TPF system to activate the IP router, reactivate the IP router after a network failure, and reactivate the IP router after an

IPL of the TPF system. The TPF system will try to keep the IP router active until you issue a ZTTCP INACTIVATE command.

When you enter a ZTTCP INACTIVATE command, a flag is set in the entry for IP router in the IP configuration record. This prevents the IP router from always becoming active, including across an IPL of the TPF system.

In a TPF production system environment where you want all IP routers active, include the ZTTCP ACTIVATE ALL command with your cycle-up procedures.

## Deleting CDLC IP Routers

Enter the ZTTCP DELETE command to delete an IP router; however, you must deactivate the IP router before it can be deleted by entering the ZTTCP INACTIVATE command.

---

## OSA-Express Connections

You must define an OSA-Express card to the processor and also define an OSA-Express connection to the TPF system before you can use OSA-Express support.

## Defining OSA-Express Cards to the Processor

Each OSA-Express card is defined to the processor in the Input/Output Configuration Program (IOCP) and includes the following information:

- The channel path identifier (CHPID) on which the OSA-Express card resides.
- The CHPID type, which is OSD for OSA-Express.
- The list of LPARs that will be using this OSA-Express card. An OSA-Express card can be dedicated to one LPAR, shared by a subset of the LPARs, or shared by all LPARs. If more than one LPAR will be sharing the card, the CHPID must be defined as SHARED.
- The SDAs that each LPAR sharing the card can use for connections to the card. The ADDRESS parameter on the IODEVICE statement specifies the first SDA and the number of sequential SDAs starting from that point. The maximum number of SDAs per LPAR is 240 divided by the number of LPARs sharing the card.
- The unit addresses of the SDAs.

**Note:** The value of the UNITADD parameter must be equal to the last 2 digits of the first SDA defined by the ADDRESS parameter. For example, if ADDRESS=(E00,080) is coded, the value of UNITADD must be 00.

The following example shows the IOCP definitions for an OSA-Express card that is shared by three LPARs called TPFTEST, TPFPROD, and MVS1. Each of these LPARs can use SDAs E00 to E4F for connections to the card.

```
CHPID PATH=F9,TYPE=OSD,SHARED,PARTITION=(TPFTEST,TPFPROD,MVS1)
```

```
CNTLUNIT CUNUMBR=F900,UNIT=OSA,PATH=(F9)
```

```
IODEVICE UNIT=OSA,ADDRESS=(E00,080),CUNUMBR=F900,UNITADD=00,  
PARTITION=(TPFTEST,TPFPROD,MVS1)
```

**Note:** You must code TYPE=OSD on the CHPID statement to use OSA-Express support on the TPF system.

## Defining OSA-Express Connections to TPF

An OSA-Express connection is defined by entering the ZOSAE command with the DEFINE and OSA parameters specified. The name of the OSA-Express connection must not have been previously defined to this processor.

Because of the number of parameters needed for OSA-Express, some are defined to the processor by first entering the ZOSAE command with the DEFINE parameter specified, while others are defined with subsequent entries of the ZOSAE command with the MODIFY parameter specified.

By entering the ZOSAE command with the MODIFY parameter specified, you can also change existing definitions. With the following exceptions, the OSA-Express connection must not be active when changing existing definitions:

- MTU size
- Gateways.

## Activating and Deactivating OSA-Express Connections

An OSA-Express connection is activated by entering the ZTTCP command with the ACTIVATE parameter specified. It takes at least 16 seconds for the activation process between the host and the OSA-Express card to be completed. This activation includes registering the real OSA IP address and any VIPAs defined to this OSA-Express connection. When the host registers its IP addresses, the OSA-Express card verifies that the real OSA IP address is unique in the network.

Activate alternate OSA-Express connections the same way before they are needed. If you do not have an active alternate OSA-Express connection and your primary OSA-Express connection fails, network traffic will be delayed for the additional 16 seconds or more it takes for the activation of the alternate OSA-Express connection to be completed. The TPF system does not automatically activate an alternate OSA-Express connection if an OSA-Express connection fails.

An OSA-Express connection can be deactivated manually by entering the ZTTCP command with the INACTIVATE parameter specified. If you enter the ZTTCP command with either the ACTIVATE or INACTIVATE parameter specified, you can specify either one connection or all connections. If you manually deactivate a connection, TPF does not swing the VIPAs to the alternate connection.

## Displaying OSA-Express Connections

To display OSA-Express connections, do one of the following:

- Enter the ZOSAE command with the DATAFLOW parameter specified to display whether there is traffic flowing across OSA-Express connections
- Enter the ZOSAE command with the DISPLAY parameter specified to display all information for one specific OSA-Express connection
- Enter the ZTTCP DISPLAY command with the ALL parameter specified to display all OSA-Express connections defined to the TPF system and the status of those connections.

## Deleting OSA-Express Connections

An OSA-Express connection is deleted by entering the ZOSAE command with the DELETE parameter specified when the OSA-Express connection is in an inactive state. The primary OSA-Express connection cannot be deleted if there is an alternate OSA-Express connection defined for it; you must delete the definition of the alternate connection before you delete the primary connection.

Deleting an OSA-Express connection deletes all the VIPAs associated with it. See *TPF Operations* for more information about the ZOSAE command.

---

## Gateways

Typically, a gateway is a router that connects networks. With an OSA-Express connection, the default gateway to be used as the first hop of a route for outbound packets is defined or modified by entering the ZOSAE command with the DEFINE or MODIFY parameter specified. A maximum of two default gateways for each OSA-Express connection can be defined, which prevents the network from having a single point of failure. You can define other gateways (besides the default gateway) by setting up IP routing table entries. See *Defining Gateways* on page 58 for more information.

## Routing Information Protocol

If you want to use VIPAs, gateways that are on the network to which the OSA-Express card connects must support Routing Information Protocol (RIP) Version 2 on the interfaces connected to that network. RIP informs the IP routers of the path to take to reach a specific TPF VIPA. The connections to other networks can use routing protocols other than RIP. The TPF system only uses RIP to broadcast VIPA information **to** the gateways.

All hosts and routers in a network can use RIP to share routing information. RIP uses User Datagram Protocol (UDP) well-known port 520. RIP messages are sent using the multicast protocol, which is a protocol that allows a single packet to be sent to all hosts and routers on the network that may need to receive the packet. However, responses to RIP requests are sent only to the node that sent the request. The TPF system sends out RIP messages to inform the network about path activations and failures. The TPF system is an IP host and not an IP router; therefore, the TPF system does not maintain routing tables or save RIP information that is received from the gateways.

---

## How Network and Processor Failures Affect VIPAs

When an OSA-Express connection fails, it causes a swing to an alternate card on that processor, if one is defined and active. If a processor fails, you can move the movable VIPAs to another processor in the loosely coupled complex.

## Swinging VIPAs to an Alternate OSA-Express Connection

If an OSA-Express connection fails and its alternate connection is active, the VIPAs automatically swing to the alternate connection on the same processor. This process, known as *swinging* VIPAs, also occurs when the alternate connection fails and the VIPAs return to their primary OSA-Express connection, if the connection is still active. You can swing VIPAs manually by entering the ZOSAE command with the SWING parameter specified. See *TPF Operations* for more information about the ZOSAE command.

## Moving VIPAs from One Processor to Another

Movable VIPAs are moved from one processor to another by:

- Processor deactivation, if directed to do so by the UVIP user exit
- The operator, by entering the ZVIPA command for load balancing traffic through the complex
- Application program, by using one of the following:

- VIPAC macro
- `tpf_vipac` C function.

### Processor Deactivation

The VIPA processor deactivation user exit, (UVIP) allows you to specify if a movable VIPA that is currently owned by a failing processor should be moved to another processor in the complex. When a processor is deactivated, UVIP is called once for each movable VIPA that meets any of the following conditions:

- The movable VIPA is owned by the deactivated processor, and the VIPA is not already in the process of being moved
- The VIPA is owned by the deactivated processor and is moving to an inactive processor
- The VIPA is being moved to the deactivated processor.

If UVIP is not coded, the VIPA is not moved.

For more information about the UVIP user exit, see *TPF System Installation Support Reference*.

### The Operator and the ZVIPA Command

Enter the ZVIPA command with the MOVE parameter specified to transfer a movable VIPA to a different processor in the same loosely coupled TPF environment. To move a VIPA, you must define it as movable to the processor where you want it moved, as described in *Defining VIPAs to an OSA-Express Connection* on page 64.

### Moving a VIPA by an Application Program

An application program can move a VIPA from one processor to another processor in the same loosely coupled TPF environment. To move a VIPA, you must define the VIPA as movable to the processor where you want it moved as described in *Defining VIPAs to an OSA-Express Connection* on page 64, and then use the VIPAC macro or the `tpf_vipac` C function to move the VIPA.

## Workload Balancing Using Movable VIPAs

Workload balancing with movable VIPAs is useful for the following:

- When taking a central processing unit (CPU) out of the complex and moving its IP addresses.
- When moving a processor out of the complex and adding another processor. Adding a CPU with movable VIPAs guarantees that the workload is moved over.
- When network traffic is not balanced.

Enter the ZVIPA command with the SUMMARY parameter specified to display system and VIPA statistics and to determine the balance of network traffic in your TPF system. If you determine that your network is not balanced, you can move the movable VIPAs from an overloaded processor to a more available one. This is shown in the following example, where:

CPU

is the processor.

PACKETS/SEC

indicates the number of messages sent and received per second as determined by the traffic for the previous minute.

## CPU UTIL

shows the immediate CPU utilization averaged over the I-streams for the designated processor.

```
User: ZVIPA SUM
System: VIPA0003I 13.03.03 OSA IP ADDRESS SUMMARY DISPLAY BEGINS

      CPU PACKETS/SEC CPU UTIL
      -----
      A           2300   76.9
      B            290   12.1
      C            650    1.2
      E            122   21.2

      END OF DISPLAY
```

The summary display shows that processor A is overloaded while processor C has a light load. To balance the network traffic, enter the ZVIPA command with the DISPLAY parameter specified to see IP activity on a per OSA IP address basis for the complex. This will help you determine which VIPAs on processor A should be moved to processor C. Before moving the VIPAs, ensure that they are defined on processor C. The following example displays OSA IP information, where:

### CPU

is the processor that currently owns the IP address.

IP is the IP address.

### TYPE

is the type of IP address, where *type* is one of the following:

#### MOVABLE

specifies an IP address that is defined as a movable VIPA.

#### STATIC

specifies an IP address that is defined as a static VIPA.

#### REAL

specifies a real OSA IP address.

### MOVING TO CPU

is the CPU to which the movable VIPA is in the process of moving.

### ACTIVE

indicates whether the IP address is active.

### PACKETS/SEC

indicates the number of messages sent and received per second as determined by the traffic for the previous minute.



```

ZOSAE DEFINE OSA-TPFAOSA1 IP-1.1.1.1 MASK-255.255.255.0 PORT-OSAPORT1 READ-E00 DATA-E02
ZOSAE MODIFY OSA-TPFAOSA1 GATEWAY1-1.1.1.3 GATEWAY2-1.1.1.4 NET-GENET
ZOSAE DEFINE OSA-TPFAOSA2 IP-2.2.2.2 MASK-255.255.255.0 PORT-OSAPORT2 READ-722 DATA-725
ZOSAE MODIFY OSA-TPFAOSA2 GATEWAY1-2.2.2.8 GATEWAY2-2.2.2.6 PRIMARY-TPFAOSA1 NET-GENET
ZOSAE ADD OSA-TPFAOSA1 VIPA-3.3.3.3 STATIC
ZOSAE ADD OSA-TPFAOSA1 VIPA-3.3.3.4

```

This environment was set up on TPFB by entering the following:

```

ZOSAE DEFINE OSA-TPFBOSA1 IP-1.1.1.2 MASK-255.255.255.0 PORT-OSAPORT1 READ-806 DATA-804
ZOSAE MODIFY OSA-TPFBOSA1 GATEWAY1-1.1.1.3 GATEWAY2-1.1.1.4 NET-GENET
ZOSAE DEFINE OSA-TPFBOSA2 IP-2.2.2.26 MASK-255.255.255.0 PORT-OSAPORT2 READ-920 DATA-922
ZOSAE MODIFY OSA-TPFBOSA2 GATEWAY1-2.2.2.8 GATEWAY2-2.2.2.6 PRIMARY-TPFBOSA1 NET-GENET
ZOSAE ADD OSA-TPFBOSA1 VIPA-3.3.3.8 STATIC
ZOSAE ADD OSA-TPFBOSA1 VIPA-3.3.3.4

```

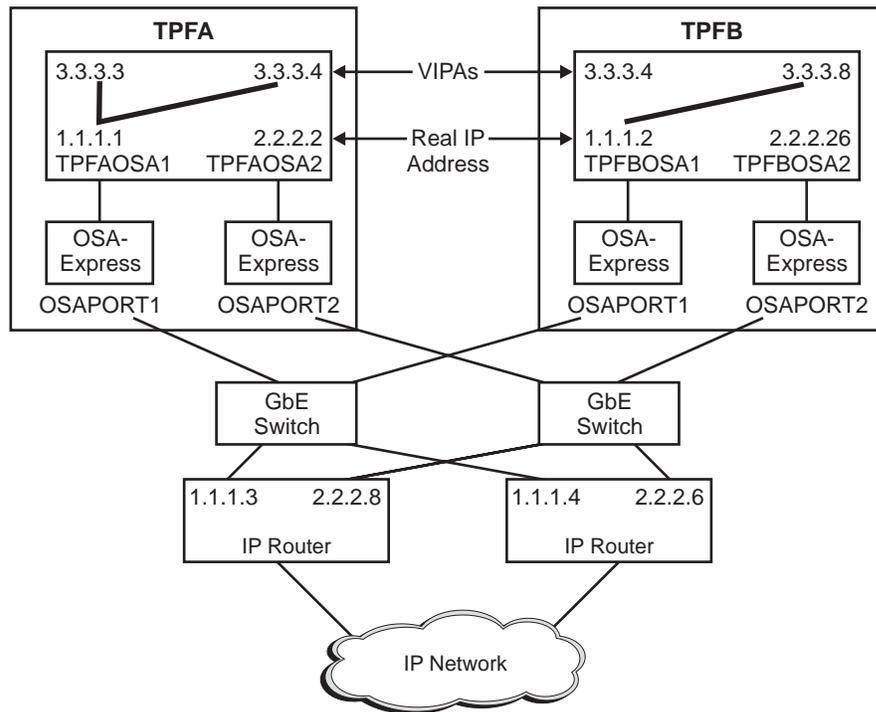


Figure 16. TCP/IP Native Stack Support Sample Computer Network Using OSA-Express with Static and Movable VIPAs

**Note:** Figure 16 shows GbE networks. If you are using Fast Ethernet (FENET) networks, do the following:

1. Replace the GbE switches with FENET hubs.
2. Code NET-FENET to replace NET-GENET to define or modify the OSA-Express connections by entering the ZOSAE command with the DEFINE or MODIFY parameter specified.

Figure 17 on page 73 shows VIPAs that have been swung from connection TPFAOSA1 to TPFAOSA2 on the TPFA processor by entering the following:

```
ZOSAE SWING OSA-TPFAOSA2
```

The ZOSAE command with the SWING parameter specified must be entered from the processor where the OSA-Express card exists.

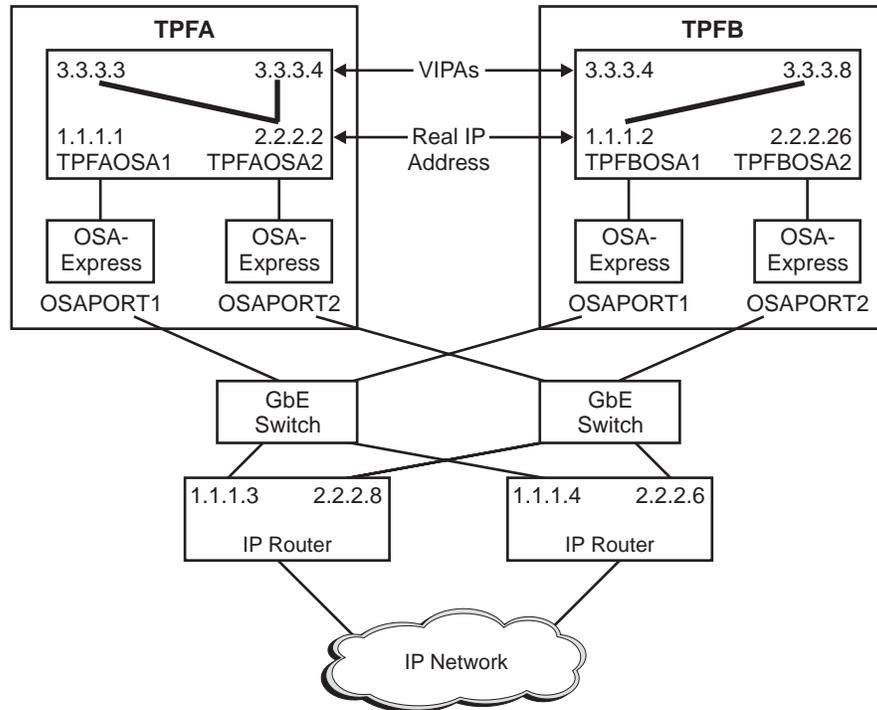


Figure 17. TCP/IP Native Stack Support Sample Computer Network Using OSA-Express with Swinging VIPAs

Figure 18 on page 74 shows a movable VIPA (3.3.3.4) that has been moved from processor TPFA to processor TPFB by entering the following:

```
ZVIPA MOVE VIPA-3.3.3.4 CPU-B
```

The ZVIPA command with the MOVE parameter specified can be issued from any processor in the loosely coupled complex.

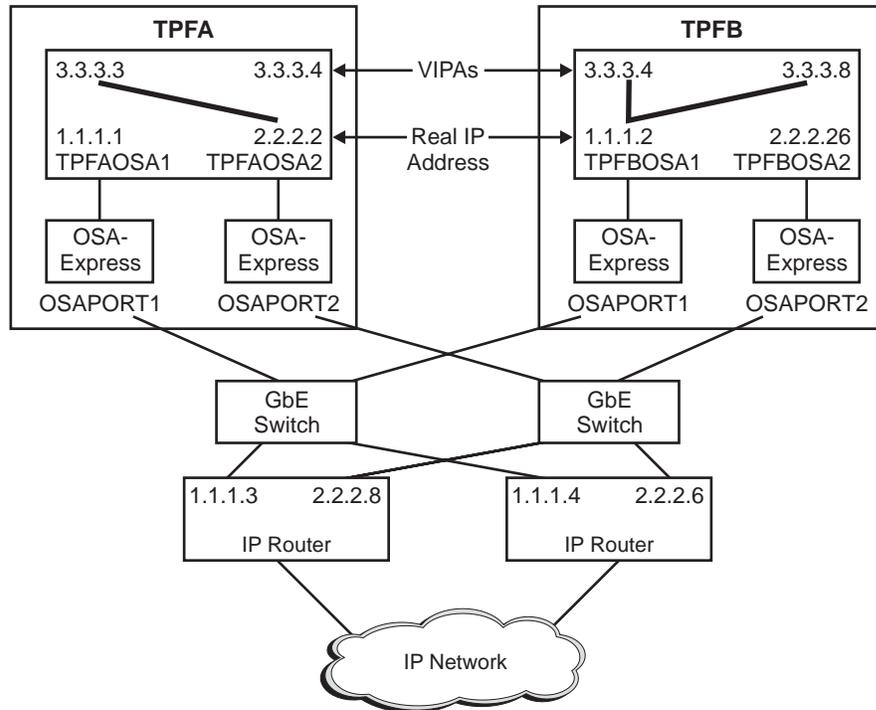


Figure 18. TCP/IP Native Stack Support Sample Computer Network Using OSA-Express with Movable VIPAs

## Managing IP Routing Table Entries

Enter the ZTRTE command to manage the IP routing table (IPRT) entries. With this command you can do the following:

- Add IPRT entries that map either a local IP address or an address of a next-hop IP router to a remote IP address or subset of remote IP addresses
- Delete IPRT entries
- Modify IPRT entries
- Display IPRT entries
- Display IPRT statistics.

See *TPF Operations* for more information about the ZTRTE command.

The remote IP address or subset of remote IP addresses must be associated with a network mask. A default network mask of 255.255.255.255 is used if the network mask is not specified when adding an IPRT entry. A network mask must consist of contiguous ones followed by contiguous zeros at the bit level, for example:

- 255.255.255.128 is X'FFFFFF80' and is a valid network mask.
- 255.255.255.64 is X'FFFFFF40' and is not a valid network mask.

Each time there is an addition, deletion, or modification of an IPRT entry, the IPRT entry is filed out. On an initial program load (IPL), the IPRT is built using these IPRT entries.

Use the ZTRTE command with the DISPLAY parameter specified to display IP routing table entries.

Use the ZTRTE command with the STAT parameter specified to display IP routing table entries including the total number of entries, the number of active entries, and the number of available entries.

---

## Displaying TCP/IP Native Stack Support

Enter the ZTTCP DISPLAY command to display information about:

- One or more IP routers, OSA-Express connections, or both. The display includes the current status and desired status of the IP routers and OSA-Express connections.
- The local IP addresses defined to this host. Specify the LOCIPS parameter to display the local IP addresses defined and their characteristics.
- Resource usage and statistical information. Specify the STATS parameter.

---

## Starting and Stopping the IP Trace Function

The IP trace function allows you to trace some or all IP packets sent and received by the TPF system. The information is saved in the main storage copy of the IP trace table and optionally written to the real-time tape to be processed offline.

Use the ZTTCP TRACE command to start or stop the IP trace function and to control its properties. You can trace:

- All IP packets
- IP packets to and from a specific TPF IP address
- IP packets to and from a specific IP router or OSA-Express card.
- Routing Information Protocol (RIP) messages, or you can select not to trace RIP messages.

Multiple traces can be active at the same time.

If an IP packet is traced, the entire IP header and protocol header (TCP header or UDP header) are saved in the trace table. If data also exists in the packet, the SIZE parameter of the ZTTCP TRACE command controls the amount of data in the packet that is also added to the trace table.

---

## Displaying IP Trace Information

Use the ZIPTR command to display the IP trace table online. The table wraps; therefore, only the most recent packets are in main storage. Use the offline IPTPRT facility to process the IP trace data that is written to the real-time tape.

The online and offline displays both have a *compact* and a *format* option. The compact option displays one packet per line. The format option displays the entire packet that was traced and translates key fields to readable text.

See Appendix F, "Using the Internet Protocol Trace Facility" on page 407 for more information.

---

## Using the Individual IP Trace Function

In addition to a systemwide IP trace table displayed by the ZIPTR command or processed offline using the IPTPRF facility, there is an individual IP trace function that allows you to trace all the packets to and from specific remote terminals in the network. The individual IP trace function is an online trace facility that allows you to define the following individual IP traces:

- All packets to and from a specific remote node
- All packets to and from a local server application
- All packets to and from a remote server application.

However, Individual IP trace support does not affect the function of the systemwide trace table.

You can define an individual IP trace by entering the ZINIP command with the DEFINE parameter specified. Assign the individual IP trace a trace name, which is used for accessing the trace for other functions (displaying the trace table, for example). The following example defines an individual IP trace:

```
ZINIP DEFINE NAME-MARDI RIP-9.117.249.58
```

You can specify the maximum number of individual IP traces you can define by using the keypoint 2 (CTK2) parameter, IPTRCNUM. You can specify the size of each individual IP trace that is defined by using the CTK2 parameter, IPTRCSIZ.

Individual IP trace support has the following capabilities:

- The individual IP trace table wraps so only the most recent packets are in main storage. Many times when tracing specific remote terminals, you want to see the initial data flows in the trace table. You can do this by entering the ZINIP command with the NOWRAP parameter specified, which allows you to stop tracing once the trace table becomes full.

To continue tracing after the trace table is full, do one of the following:

- Enter the ZINIP command with the RESET parameter specified to clear the table and to continue tracing.
  - Enter the ZINIP command with the MODIFY and WRAP parameters specified to continue the tracing.
- The individual IP trace function allows you to pause an individual IP trace so you can examine what has been traced to a certain point. You can resume the trace by entering the ZINIP command with the RESUME parameter specified.
  - Entering the ZINIP command with the SIZE parameter specified controls the amount of data in the packet that is added to the trace table.
  - Entering the ZINIP command with the SUMMARY parameter specified allows you to see what individual IP traces are currently defined and their status, as shown in the following example.

```
User: ZINIP SUMMARY

System: INIP0005I 11.43.04 INDIVIDUAL IP TRACE SUMMARY
NAME REMOTE IP PORT SIZE WRAP PAUSED STATUS
-----
MARDI 9.117.249.058 3840 NO NO EMPTY
MQSERV 1414 3840 YES NO EMPTY
END OF DISPLAY
```

See *TPF Operations* for more information about the ZINIP command.

---

## Displaying Individual IP Trace Tables

Enter the ZINIP command with the DISPLAY parameter specified to display the packets in an individual IP trace table as formatted or nonformatted online output. Specifying the FORMAT parameter displays the entire packet that was traced and translates key fields to readable text, while not specifying the FORMAT parameter displays one packet per line.

The following example shows a formatted display of packets in an individual IP trace table:

```
User: ZINIP DISP NAME-MARDI 3 FORMAT
System: INIP0007I 13.55.27 INDIVIDUAL IP FORMATTED TRACE MARDI DISPLAY
RWI-32 IPCCW-01 SOURCE IP-9.117.249.58 DEST IP-9.117.249.59 LEN-1040
TOD-B5553A79A25E2700 PROTOCOL-06 (TCP) SOURCE PORT-1025 DEST PORT-9999
SEQ-2786811134 ACK-2788236297 WINDOW-8192 URGENT OFFSET-0
TCP FLAG BYTE-18 (ACK, PSH)
IP HEADER 45000410 D0B30000 3B06A5D4 0975F93A 0975F93B
TCP HEADER 0401270F A61B5CFE A6311C09 50182000 07D00000
RWI-32 IPCCW-01 SOURCE IP-9.117.249.58 DEST IP-9.117.249.59 LEN-1040
TOD-B5553A7A06FB6E87 PROTOCOL-06 (TCP) SOURCE PORT-1024 DEST PORT-9999
SEQ-1063223070 ACK-1064793518 WINDOW-8192 URGENT OFFSET-0
TCP FLAG BYTE-18 (ACK, PSH)
IP HEADER 45000410 D0B40000 3B06A5D3 0975F93A 0975F93B
TCP HEADER 0400270F 3F5F7F1E 3F7775AE 50182000 59820000
RWI-32 IPCCW-01 SOURCE IP-9.117.249.58 DEST IP-9.117.249.59 LEN-1040
TOD-B5553A7A06FB6E87 PROTOCOL-06 (TCP) SOURCE PORT-1025 DEST PORT-9999
SEQ-2786812134 ACK-2788237297 WINDOW-8192 URGENT OFFSET-0
TCP FLAG BYTE-18 (ACK, PSH)
IP HEADER 45000410 D0B50000 3B06A5D2 0975F93A 0975F93B
TCP HEADER 0401270F A61B60E6 A6311FF1 50182000 00000000
119 ENTRIES IN IP TRACE TABLE
```

See *TPF Operations* for more information about the ZINIP command.

---

## Deactivating Sockets

Use the ZTTCP INACTIVATE command, specifying the SOCKETS parameter to deactivate all of the sockets on a TPF host. When a TPF host processor is being deactivated, deactivate all sockets before cycling down below CRAS state.

When the TPF system is cycled below CRAS state, all sockets are cleaned up. For each connected TCP socket, a reset (RST) message is built to inform the remote end that the connection is being cleaned up. However, it is not guaranteed that the reset messages will be sent out before the TPF system cycles down. For this reason, enter the ZTTCP INACTIVATE command with the SOCKETS parameter specified before you cycle down your TPF system.

If the TPF system cleans up TCP sockets internally without notifying the remote partners (for example, during an unplanned TPF outage), one of two actions will occur:

1. The remote end will send data while the TPF system is IPLing. No acknowledgments will be received, causing the remote end to detect the failure of the TPF system and clean up the sockets on the remote end.
2. The TPF system is cycled up after the IPL and the remote end cleans up the socket when it sends the next message. The first message sent to the TPF system on any of these old sockets will cause a reset message to be sent to the remote end causing the remote end to clean up the old socket.

Enter the ZSOCK command with the INACT parameter specified to deactivate a specific socket or all sockets that have matching values for the selection criteria. The selection criteria can be any of the following:

- Local TPF IP address
- Local port number
- Remote IP address
- Remote port number
- Socket protocol.

See *TPF Operations* for more information about the ZSOCK command.

---

## Displaying Socket Control Block Information

Enter the ZSOCK command with the DISPLAY parameter specified to display a socket descriptor or a summary table of all socket descriptors matching specified selection criteria. The selection criteria can be any of the following:

- Local TPF IP address
- Local port number
- Remote IP address
- Remote port number
- Socket protocol.

Enter the ZSOCK command with the FORMAT parameter specified if you want a formatted version of the socket control block information.

Enter the ZSOCK command with the CONVERT parameter specified to find a specified socket descriptor.

Enter the ZSOCK command with the DATAFLOW parameter specified to see if any data is flowing on the specified socket.

See *TPF Operations* for more information about the ZSOCK command.

---

## Socket Application Design Considerations

This chapter discusses:

- Sharing of sockets by multiple entry control blocks (ECBs)
- Using existing socket applications
- New socket options supported by TCP/IP native stack support
- A new socket application programming interface (API) call supported by TCP/IP native stack support.

---

### Sharing Sockets

Sockets that use TCP/IP native stack support are not limited to a single ECB, subsystem, or I-stream. For example, an ECB running in subsystem ABC on I-stream 2 creates a socket. The file descriptor of this socket is then saved in a user table and the ECB exits. Next, an ECB running in subsystem XYZ on I-stream 4 retrieves the file descriptor of the socket from the user table and issues a socket API call. This is allowed.

Multiple ECBs can issue socket API calls for the same socket at the same time. For example, one ECB can issue a read API call for a socket at the same time that another ECB issues a send API call for this socket. You can even have multiple ECBs issue send API calls for the socket at the same time. The TCP/IP native stack support code handles the necessary serialization. You can issue any combination of socket API calls for a given socket with the following restrictions:

1. Only one `activate_on_receipt` API call can be pending. If an `activate_on_receipt` API call is issued for a socket, that `activate_on_receipt` call must be completed successfully before you issue the next `activate_on_receipt` call for this socket.
2. An `activate_on_receipt` API call cannot be issued if a `read`, `recv`, or `recvfrom` API call is in progress for the socket. These operations are mutually exclusive.
3. Only one `activate_on_accept` API call can be pending. If an `activate_on_accept` API call is issued for a socket, that `activate_on_accept` call must be completed successfully before you issue the next `activate_on_accept` call for this socket.
4. An `activate_on_accept` API call cannot be issued if an `accept` API call is in progress for the socket. These operations are mutually exclusive.

---

### Using Existing Socket Applications

Socket applications that were developed and run by using TCP/IP offload support will run using TCP/IP native stack support. The new code is designed to be backward compatible. There are additional socket options and a new API call supported now, but your application has to code these new options for them to work.

You can use the newly supported socket options to make your application more effective. For example, if your existing applications previously needed a timeout capability when reading data, you had to code a `select` for read API function before each read API function. With TCP/IP native stack support, the read function now has a timeout capability; therefore, you can delete the `select` for read calls in your application and use the read timeout capability instead. This decreases the number

of socket API calls issued, can cause data to be delivered faster to your application, and can decrease the number of times the data is copied.

Server applications that needed the timeout capability were also forced to code a select for read API function before their accept call. The select call is no longer needed for this because accept now has a timeout capability.

Another example would be for TCP applications that receive large messages. Your existing applications probably issue multiple read API calls and receive the message a small piece at a time, and the application has to put the message back together. Because the receive low-water mark socket option is now supported, you can have the TCP/IP native stack support code put the pieces of the message back together and then deliver the entire message to your application on a single read API call. This again reduces the number of socket API calls issued.

---

## New Socket Options Supported

TCP/IP native stack support supports socket options on the setsockopt API call that were not previously supported. These new options give application programmers more capability and can improve application productivity. The new options are:

### **SO\_RCVBUF**

Sets the receive buffer size for the socket.

### **SO\_RCVLOWAT**

Sets the receive low-water mark for the socket.

### **SO\_RCVTIMEO**

Sets the receive timeout value for the socket.

### **SO\_SNDBUF**

Sets the send buffer size for the socket.

### **SO\_SNDLOWAT**

Sets the send low-water mark for the socket.

### **SO\_SNDTIMEO**

Sets the send timeout value for the socket.

## Send Buffer and Receive Buffer Sizes

The SO\_RCVBUF option sets the receive buffer size and the SO\_SNDBUF option sets the send buffer size. These options allow you to limit the amount of IP message table (IPMT) storage used by a given socket and are a primary flow control mechanism.

For TCP sockets, each packet that is sent includes a window size indicating how much more data the remote end is allowed to send. The TPF system sets the window size to the amount of available space in the receive buffer of the socket. When the TPF application can process the data faster than it is sent, flow control is not really an issue. However, if the data arrives faster than the TPF application can process it, the rate at which the remote end sends data needs to be controlled.

Set the receive buffer size to a value just large enough that the TPF application has data to process, which means that when the TPF application issues a read API call, there is data available. The goal is to read the data fast enough so the next piece of data is always ready when the TPF application requests (reads) the next message. Reading data too fast can cause the IPMT to become full because many messages will be queued while waiting for the TPF application to process them.

For UDP sockets, controlling the send and receive buffer sizes is the only flow control mechanism available unless you include application-level acknowledgments into your application design.

## Timeouts

Before TCP/IP native stack support, `select` was the only socket API call that had a timeout capability. For all other socket API calls, if the socket is running in blocking mode, control would not be returned to the application until the operation was completed successfully, which could take minutes or even hours on a read API call (for example, if the remote end has no data to send).

The `SO_RCVTIMEO` option defines the receive timeout value, which is how long the TPF system waits for certain socket API calls to be completed before the operation times out. The `SO_RCVTIMEO` value is used for socket API calls that are waiting for data to arrive. These include `read`, `recv`, `recvfrom`, `activate_on_receipt`, `activate_on_receipt_with_length`, `accept`, `activate_on_accept`, and `connect`. For example, assume the `SO_RCVTIMEO` value for a socket is 5 seconds. If a read API call is issued for the socket and no data arrives in 5 seconds, control will be passed back to the application with a return code indicating that the operation timed out.

The `SO_SNDTIMEO` option defines the send timeout value, which is how long the TPF system waits for send-type API calls to be completed before the operation times out. These include `send`, `sendto`, `write`, and `writetv`. A send-type operation is blocked when there is not enough room in the send buffer of the socket to build the packets for the new data passed on this send-type API call. For TCP sockets, this can happen when you send data faster than the remote end can process it.

The default for both the `SO_RCVTIMEO` and `SO_SNDTIMEO` values is 0, which means do not time out. If your application does not change these values, the code will operate as it did before TCP/IP native stack support. If your application does change the `SO_RCVTIMEO` or `SO_SNDTIMEO` option to a nonzero value, the application must be prepared to get back a timeout return code.

## Low-Water Marks

When a read API call is issued for a TCP socket, the application specifies the maximum amount of data to read. If, for example, the maximum amount of data to read is  $x$ , the application could receive  $x$  or fewer bytes of data on its read API call because TCP does not have any concept of a message. If the application wants exactly  $x$  bytes of data, it often has to issue multiple read calls because the data is received in multiple packets from the network. For example, the TPF application issues a read call specifying a maximum length of 10 000 bytes. The remote application sends 10 000 bytes of data; however, the data is sent as five 2000-byte packets. When the first packet arrives, the read call is completed and the application is passed 2000 bytes. The application must then issue another read call, specifying a maximum of 8000 bytes this time to read in the remaining data. Depending on the timing of when the packets arrive, the application might have to issue five read calls to read in all of the data.

The `SO_RCVLOWAT` option allows a TCP application to indicate the minimum amount of data to pass to the application. Using the same example, the application would set the `SO_RCVLOWAT` value to 10 000 and issue one read call. The TCP/IP native stack support code will wait for 10 000 bytes of data to arrive and then pass all 10 000 bytes to the application. This reduces the number of socket API calls issued.

A socket application uses the `select` for write API to see if a socket is writable. If there is at least 1 byte of available space in the send buffer of the socket, the socket is considered writable. If an application has a 4000-byte message to send, it really wants to know if there are at least 4000-bytes available in the send buffer of the socket. The `SO_SNDLOWAT` option allows the application to set the minimum amount of space that must be available in the send buffer before a `select` for write operation will consider the socket to be writable.

The default values for the `SO_RCVLOWAT` and `SO_SNDLOWAT` options are both set to 1. This means that if your application does not change these values, the code will operate as it did before TCP/IP native stack support.

## **activate\_on\_accept API**

The new `activate_on_accept` API call is available for sockets that use TCP/IP native stack support. This API call performs the same function as the `accept` API call, but does so in a way that no ECBs are tied up while waiting for remote clients to be connected.

When a TCP server application issues an `accept` API call, the ECB is suspended until a remote client connects or until the `accept` operation times out (if the `SO_RCVTIMEO` option is enabled on the listener socket). In addition, many server designs have an ECB in a loop issuing `accept`, passing the connection to a new ECB and then the original ECB issues another `accept` call. This results in long-running ECBs. If you have many TCP server applications active, you can end up with many suspended ECBs as well, all waiting for the `accept` call to be completed successfully.

The `activate_on_accept` API function allows a TCP server application to indicate which TPF program to activate when a remote client connects. Control is always returned immediately to the ECB that issued `activate_on_accept`, allowing it to exit. While we are waiting for a remote client to be connected, there are no ECBs tied up in the TPF system. When a remote client does connect, a new ECB is created and the specified TPF program is activated.

The `activate_on_accept` API allows you to specify the I-stream on which to activate the TPF program in the new ECB. The default is 0, which means select the least busy I-stream. This enables you to load balance instances of your server application by using the TPF I-stream scheduler logic.

---

## **Local Sockets**

Local sockets is a form of loopback processing that allows you to test a socket client and server application running in the same TPF system without using or needing a real network.

When an IP packet is built by the TPF system, the destination IP address in the packet is examined and, if it is a local IP address of this TPF host, the packet is placed on the input list to make it look like the packet was received from the network. When this occurs the packet is not added to the IP output queue to be sent to the network.

To use local sockets, you do not need to have any IP routers active or even defined to your TPF system.

Local sockets can be used in TPF test systems to test new socket applications. Local sockets can also be used in a TPF production system environment. For example, if you have a socket client application on your TPF system that sends messages to server applications that reside in different locations (one of which is in the same TPF system as the client), the client application can use the same socket interface to communicate with remote servers as well as the local server.



---

# Simple Network Management Protocol Agent Support

This chapter describes Simple Network Management Protocol (SNMP) agent support for the TPF system.

---

## SNMP Overview

As Transmission Control Protocol/Internet Protocol (TCP/IP) networks have become increasingly diverse and complex with many different types of devices and network nodes connected to them, it has become more challenging to manage them. SNMP is a standard protocol that was developed to provide an effective way to centralize the management of TCP/IP networks. SNMP consists of three major components that communicate with each other to manage and monitor TCP/IP networks:

- SNMP managers
- SNMP agents
- Management Information Bases (MIBs).

SNMP is defined by a series of Request for Comments (RFC) documents that describe the specifications for network management including the protocol itself, the definition of data structures, and associated concepts. The architecture for standard TCP/IP network management protocols is defined by the following RFC documents:

- RFC 1155 *Structure and Identification of Management Information for TCP/IP-based internets*
- RFC 1157 *A Simple Network Management Protocol (SNMP)*
- RFC 1213 *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*
- RFC 2233 *The Interfaces Group MIB using SMIv2.*

Go to <http://www.ietf.org> for more information about these RFCs and any related extensions.

## SNMP Manager

The SNMP manager consists of a set of applications that monitor all the other nodes (SNMP agents) in the network and send requests for information to SNMP agents in the network. The SNMP manager requests TCP/IP information from the SNMP agents in the network by requesting the variables from each agent's MIB database. The manager also receives and processes unsolicited messages from the agents, which indicate that a significant event has occurred at a specific network node. These unsolicited messages are called *traps*.

## SNMP Agent

An SNMP agent is an entity on the network that supplies the SNMP manager with TCP/IP network information. An SNMP agent maintains the MIB database of TCP/IP network information pertaining to its network node. The SNMP agent responds to requests from the SNMP manager for information from its MIB and sends traps to SNMP managers when significant events occur.

## Management Information Base (MIB)

The MIB is a database of variables and their associated values that is maintained by the SNMP agent about the node on which it resides. The SNMP manager solicits

information from the MIB of the SNMP agent. Each piece of information in the MIB database is called an *MIB variable*. See Appendix G, “Management Information Base Variables” on page 423 for a list of all the MIB-II variables supported by the TPF system.

Use the ZSNMP command to display the MIB variables from the TPF system. You can also save the display information to a file. See *TPF Operations* for more information about the ZSNMP command.

## Interaction between SNMP Components

Figure 19 shows a high-level view of the interaction between the three major components of SNMP. Specifically, this figure shows an SNMP agent with its MIB database of TCP/IP networking information being monitored by an SNMP manager. Queries can be issued from an SNMP manager to obtain statistical information about TCP/IP native stack support networks.

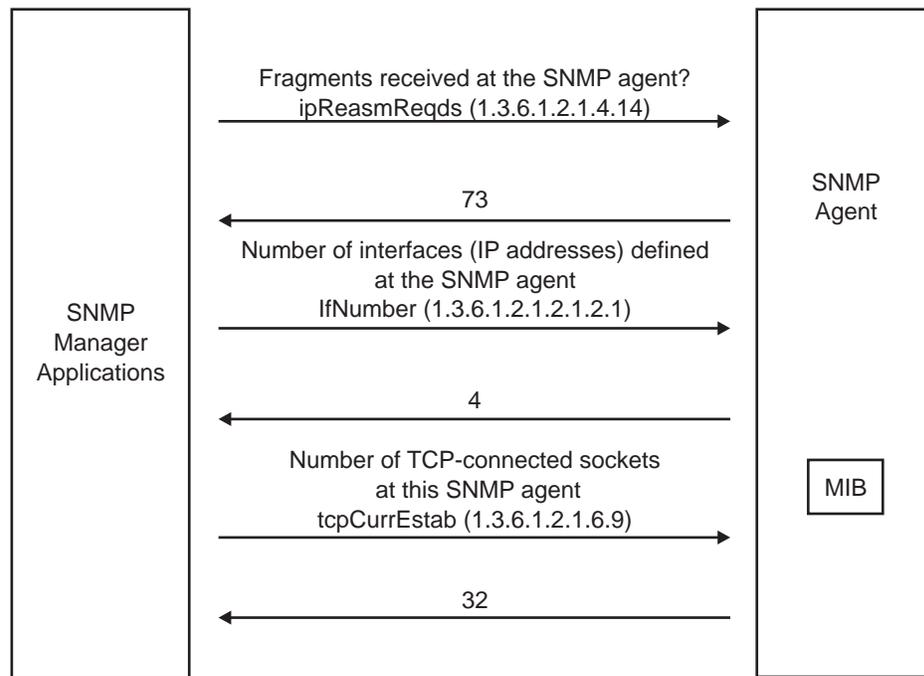


Figure 19. Interaction between SNMP Components

Figure 20 on page 87 shows an example of an SNMP trap message being sent by an SNMP agent to the SNMP managers. When the SNMP agent system loses one of its TCP/IP links, it notifies the SNMP managers that a link went down.

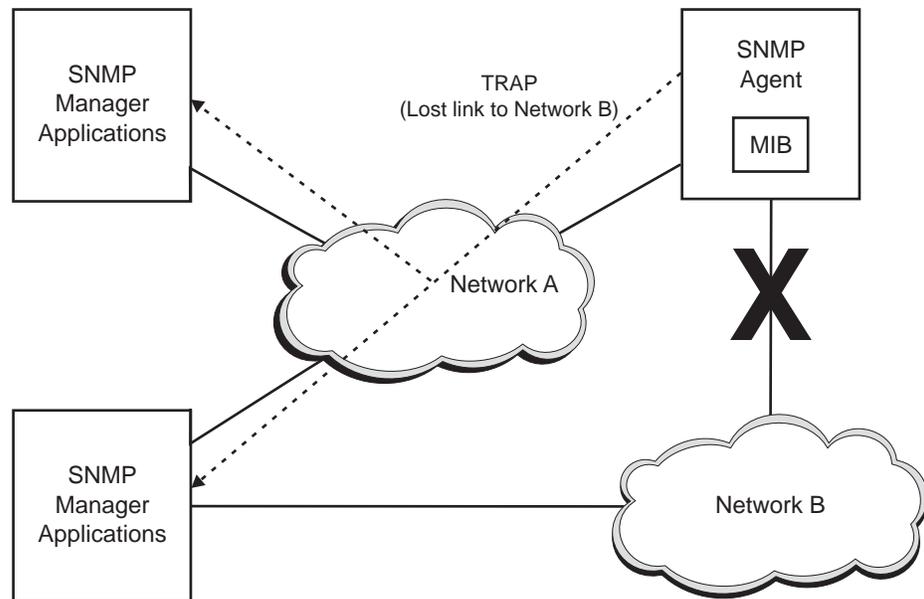


Figure 20. SNMP Agent Sending a Trap Message

---

## Protocol Data Units (PDUs)

Each SNMP message contains a protocol data unit (PDU). These SNMP PDUs are used for communication between SNMP managers and SNMP agents. The SNMP Version 1 architecture defines the following types of PDUs that flow between SNMP managers and SNMP agents:

### GETREQUEST PDU

Sent by the SNMP manager to retrieve one or more requested MIB variables specified in the PDU.

### GETNEXTREQUEST PDU

Sent by the SNMP manager to retrieve the next MIB variable that is specified in the PDU. You can have multiple requests in the PDU. This PDU is primarily used by the SNMP manager to walk through the SNMP agent MIB.

### SETREQUEST PDU

Sent by the SNMP manager to set one or more MIB variables specified in the PDU with the value specified in the PDU.

### GETRESPONSE PDU

Sent by the SNMP agent in response to a GETREQUEST, GETNEXTREQUEST, or SETREQUEST PDU.

### TRAP PDU

An unsolicited message sent by the SNMP agent to notify the SNMP manager about a significant event that occurred in the agent.

## Structure and Fields of SNMP PDUs

There are many types of structures and fields contained in the PDUs supported by the TPF system. Table 2 on page 88 shows the format of the SNMP PDUs.

Table 2. GETREQUEST, GETNEXTREQUEST, SETREQUEST, GETRESPONSE PDU Format

Version	Community name	PDU type	Request ID	Error status	Error index	Variable binding list
---------	----------------	----------	------------	--------------	-------------	-----------------------

**Version**

The version of the SNMP message. The TPF system supports version 1.

**Community name**

An ASCII display string of the name of the community from where the PDU originated. This value can be up to 255 characters in length.

**PDU type**

The type of PDU contained by the SNMP message. PDU type can be one of the following:

- GETREQUEST
- GETNEXTREQUEST
- SETREQUEST
- GETRESPONSE

**Request ID**

A unique number that is used to distinguish between different requests and to associate them with the corresponding response.

**Error status**

Used to indicate that an error occurred while the agent was processing a request.

**Error index**

Used to provide additional information by identifying which variable in the list caused an error.

**Variable binding list**

A simple list of variable bindings, which are pairings of the names of MIB variables with their corresponding values.

Table 3 shows the format of the trap PDUs.

Table 3. Trap PDU Format

Version	Community name	PDU type	Enterprise object identifier	Network address	Trap type	Specific trap type	Time stamp	Variable binding list
---------	----------------	----------	------------------------------	-----------------	-----------	--------------------	------------	-----------------------

**PDU type**

The type of PDU contained by the SNMP message; in this case, a trap PDU.

**Enterprise object identifier**

The unique identifier of the SNMP agent that is sending the trap. This value can be up to 255 characters in length.

**Network address**

The default IP address of the SNMP agent that is sending the trap.

**Trap type**

The type of trap PDU being sent. The following trap values can be defined:

- Authentication failure
- Coldstart
- EggNeighborLoss
- Enterprise-specific

- Linkdown
- Linkup
- Warmstart

For more information, see “SNMP Traps” on page 93.

**Specific trap type**

A user-defined value for an enterprise-specific trap.

**Time stamp**

The system up time, in hundredths of a second, for the system generating the trap.

All SNMP PDUs can contain a list of variable bindings. Table 4 shows the format of the variable binding.

*Table 4. Variable Binding Format*

Variable binding type:	Length of the variable binding	Type of object identifier	Length of object identifier	Value of object identifier	Type of value for this object identifier	Length of value for this MIB variable	Value for this MIB variable
SEQUENCE_OF							

---

## TPF SNMP Agent Support

TPF support for the SNMP agent is a User Datagram Protocol (UDP) server that responds to requests received from SNMP managers. The TPF Internet daemon monitors the network for SNMP requests destined for the SNMP agent server. SNMP requires the agent to be bound to well-known port 161.

The TPF SNMP agent supports SNMP Version 1. The TPF SNMP agent supports and maintains the Management Information Base-II (MIB-II) database of network statistics and data for TCP/IP native stack support networks. The MIB-II database splits the MIB variables into groups, and only the groups pertaining to TPF are supported by TPF. You must define some MIB variables in the SNMP configuration file, which resides on the TPF file system as `/etc/snmp.cfg`. See “Creating the SNMP Configuration File” on page 96 for more information about creating the `/etc/snmp.cfg` SNMP configuration file, and see Appendix G, “Management Information Base Variables” on page 423 for a detailed list of all the MIB-II groups and variables that the TPF system supports.

In addition to the predefined MIB-II variables supported by the TPF system, you can provide your own enterprise-specific MIB variables. These enterprise-specific MIB variables allow you to track information that is of specific interest to you in managing your network. A user exit, UMIB, is provided to allow you to process these enterprise-specific MIB requests. See “User MIB Variables” on page 93 for more information about enterprise-specific MIB variables. See *TPF System Installation Support Reference* for information about the UMIB user exit.

The TPF system can also send the following predefined SNMP trap PDUs:

- Authentication failure trap
- Coldstart trap
- Linkdown trap
- Linkup trap

**Note:** In TPF SNMP agent support, the Warmstart and EgpNeighborLoss traps are not applicable and are not supported. See “SNMP Traps” on page 93 for more information about SNMP traps.

TPF SNMP agent support also allows you to send your own enterprise-specific trap PDUs by using the ITRPC macro or the `tpf_itrpc` C/C++ function. You can send enterprise-specific traps to one or more SNMP managers defined in the `/etc/snmp.cfg` SNMP configuration file. If desired, you can disable SNMP traps, which prevents all SNMP traps from being sent to the network. See *TPF General Macros* for more information about the ITRPC macro and see the *TPF C/C++ Language Support User's Guide* for more information about the `tpf_itrpc` C/C++ function.

The TPF system processes and responds to all GETREQUEST and GETNEXTREQUEST PDUs received. All SETREQUEST PDUs received by TPF are rejected with a NOSUCHNAME error code because of the security risk involved when allowing MIB variables in TPF to be set by a remote SNMP manager.

## Implementing Management Information Base-II (MIB-II)

The TPF system uses the MIB-II database to provide its MIB variables and maintains it in a core memory table. Core memory for the MIB is allocated during restart and reinitialized after an IPL. Each processor in a loosely coupled complex has and maintains its own MIB. See Appendix G, "Management Information Base Variables" on page 423 for a list of all the MIB-II variables supported by the TPF system.

The MIB-II database consists of many units of information that provide performance and statistical information about the TCP/IP networks connected to the TPF system. The TPF system updates and maintains the MIB-II variables for networks established with TPF TCP/IP native stack support. Each MIB-II variable has a unique name in Abstract Syntax Notation One (ASN.1) format called an *object identifier*, or *object ID*, that is used when a request PDU is received from the network to retrieve the MIB-II variable information. The MIB-II database defines the following SNMP groups:

### **Address translation group**

Not supported by the TPF system because TPF does not translate addresses.

### **EGP group**

Not supported by the TPF system because TPF is not a gateway.

### **ICMP group**

Contains statistical and error information related to the Internet Control Message Protocol (ICMP) layer.

### **Interfaces group**

Contains statistical information about Internet Protocol (IP) interfaces or addresses.

### **IP group**

Contains statistical and error information related to the IP layer.

### **SNMP group**

Contains statistical and error information related to SNMP.

### **System group**

Contains administrative information about the TPF system.

### **TCP group**

Contains statistical and error information related to the Transmission Control Protocol (TCP) layer.

### **Transmission group**

Not supported by the TPF system because it does not apply to TPF.

### UDP group

Contains statistical and error information related to the UDP layer.

Some of these groups contain table variables, and the variables in the table pertain to multiple entities. For example, the interfaces group mentioned previously contains an interface table with statistical information for each interface or IP address in the TPF system.

---

## Processing SNMP Requests

You communicate between the SNMP manager and SNMP agent through the exchange of messages, each within a single UDP packet. The TPF system supports a maximum SNMP packet size of 548 bytes (excluding the IP and UDP headers). One SNMP agent server can be started per processor in a loosely coupled TPF complex by using the Internet daemon to monitor the network for SNMP packets.

As shown in the following table, an SNMP message consists of a version identifier, an SNMP community name, and a PDU.

Table 5. SNMP Network Management Protocol Packet

SNMP version	Community name	PDU=PDU type, MIB variable 1, MIB variable 2, MIB variable 3, ....MIB variable <i>n</i>
--------------	----------------	-----------------------------------------------------------------------------------------

All units of data in an SNMP message are encoded using a subset of the basic encoding rules (BER). See ISO 8825 *Part 1: Basic Encoding Rules* for more information. Go to <http://www.iso.ch/> to view ISO 8825.

## Message Processing

The processing of an SNMP message begins when TPF receives an SNMP packet on well-known port 161. Message processing then continues as follows:

1. The Internet daemon activates the SNMP agent when a packet is received from the network.
2. The SNMP agent parser examines the SNMP packet for required fields. Each SNMP packet that the TPF system receives must contain the SNMP version, community name, and PDU. If the message that is received cannot be parsed because it is not in the correct format, the message is discarded.
3. The version field of the SNMP packet is validated for SNMP Version 1. Any packet that is not from SNMP Version 1 is discarded by the TPF system.
4. The TPF system passes the community name received in the SNMP message and the remote IP address of the SNMP manager that sent the SNMP request to the UCOM user exit. The UCOM user exit validates the SNMP manager from which the message was received. If an SNMP agent receives a request from an SNMP manager that is not authorized, the request from that manager is rejected and one of the following actions occurs:
  - The SNMP packet is discarded.
  - The SNMP packet is discarded and a trap message is sent to all SNMP managers defined in the SNMP configuration file to notify them that there has been an authentication failure.

You can select one of these actions by specifying the appropriate return code when coding the UCOM user exit. See “Coding the UCOM and UMIB User Exits” on page 98 and *TPF System Installation Support Reference* for more information about coding the UCOM user exit.

5. Once the community name is validated, the PDU section of the packet is parsed to determine the type of PDU request that was received and which MIB variables are being requested.
6. If the PDU is a GETREQUEST or GETNEXTREQUEST PDU, the TPF system extracts the object identifiers from the PDU to determine which MIB variables to retrieve. The TPF system then attempts to retrieve the values for each requested variable from the MIB. Multiple variables may be requested in a single PDU.

**Note:** If the PDU is a SETREQUEST PDU, the TPF system rejects SETREQUEST by sending a GETRESPONSE PDU with an error status code of NOSUCHNAME.

7. If it is determined that the requested object identifier is not part of the system-controlled MIB, the request is sent to the UMIB user exit to retrieve your own enterprise-specific MIB variables. The object identifier is provided as input to the UMIB user exit, so the user exit can do Step 1 or Step 2.

Step 1:

- Retrieve the MIB variable
- Correctly encode the MIB variable and the value
- Return the MIB variable to the SNMP agent parser.

Step 2:

- Return an error that causes the SNMP agent to reject the request with an error status code of NOSUCHNAME if the enterprise-specific MIB variable cannot be retrieved by the UMIB user exit.

See “User MIB Variables” on page 93 for more information.

8. For each variable (object identifier) requested, its value is retrieved from either the system-controlled MIB or from the UMIB user exit and a *variable binding* is built. A variable binding is the unique object identifier of an encoded variable followed by its encoded value.

The following example shows a variable binding requesting variable *ifNumber* from the TPF system.

300B	06072B06010201020100	0500
Variable Binding	Encoded object ID for ifNumber	NULL Value
Length encoded as type SEQUENCE_OF		

The following example shows a variable binding response of variable *ifNumber*.

300D	06072B06010201020100	020104
Variable Binding	Encoded object ID for ifNumber	4 interfaces encoded as an integer
Length encoded as type SEQUENCE_OF		

An SNMP manager can make a request for the value of more than one MIB variable in an SNMP packet. The TPF system retrieves the value for each requested MIB variable and builds a variable binding. All the variable bindings are then packaged together by encoding them as a *variable binding list*, which is a

simple list of variable names and their corresponding values. The variable binding list is included in the PDU section of the SNMP response packet and sent to the SNMP manager.

**Note:** If any variables cannot be retrieved by either the TPF system or by the UMIB user exit, the entire variable binding list is rejected.

## User MIB Variables

As stated previously, the SNMP architecture allows you to have your own enterprise-specific MIB variables through the use of the UMIB user exit. When an SNMP request is received by the TPF system and the requested MIB variable is not one of the SNMP system MIB variables, the UMIB user exit is called to provide the value for the specified variable or to indicate that the specified variable is not an enterprise-specific MIB variable. If you or the TPF system could not retrieve any of the requested MIB variables, the SNMP request is rejected.

Each MIB variable has an object identifier, which is BER-encoded and passed as input to the UMIB user exit. If the specified object identifier is known to you and is coded in the UMIB user exit, the UMIB user exit returns the BER-encoded object identifier and value for the requested MIB variable. The `tpf_snmp_BER_encode` C/C++ function allows you to encode a subset of the MIB variable types. See the *TPF C/C++ Language Support User's Guide* for more information about the `tpf_snmp_BER_encode` C/C++ function. See ISO 8825 *Part 1: Basic Encoding Rules* for more information about BER encoding. Go to <http://www.iso.ch/> to view ISO 8825.

The UMIB user exit retrieves enterprise-specific MIB variables. You must maintain these variables and allocate and maintain your own storage for them. The SNMP architecture has created the `iso.org.dod.internet.private.enterprises` (1.3.6.1.4.1) portion of the SNMP hierarchical tree specifically for enterprise-specific MIB variables. Object identifiers that the SNMP architecture has set aside for the various system-controlled SNMP groups cannot be used for enterprise-specific purposes.

See “Coding the UCOM and UMIB User Exits” on page 98 and *TPF System Installation Support Reference* for more information about the UMIB user exit.

---

## SNMP Traps

The SNMP architecture defines a set of system traps, which are the unsolicited messages that are sent out when a significant event occurs on the TCP/IP network node on which the SNMP agent resides. You have the ability to generate your own enterprise-specific traps.

The TPF system sends traps to remote SNMP managers on well-known port 162. You can specify to which managers to send traps by coding them in the `/etc/snmp.cfg` SNMP configuration file. Traps can be suppressed by specifying a value of NONE on the TRAPIP keyword in the `/etc/snmp.cfg` SNMP configuration file. The TPF system can send out the following types of SNMP traps:

### Authentication failure trap

Generated by the TPF system when an SNMP message is received and verification of the SNMP manager failed. By specifying the appropriate return code when the UCOM user exit is called to validate the SNMP manager, you can choose whether or not the TPF system should send out an authentication failure trap.

### Coldstart trap

Generated by the TPF system when you first cycle-up after an IPL when the MIB database is initialized. At this time, notification is sent to the SNMP managers.

### Enterprise-specific trap

You have the ability to send enterprise-specific traps by issuing the ITRPC macro or the `tpf_itrpc` C/C++ function. When you send an enterprise-specific trap by issuing the ITRPC macro or `tpf_itrpc` C/C++ function, a variable binding list can be supplied to send with the trap. You must correctly encode the variable binding list when issuing the C/C++ function. See *TPF General Macros* for more information about the ITRPC macro and see the *TPF C/C++ Language Support User's Guide* for more information about the `tpf_itrpc` C/C++ function.

### Linkdown trap

Generated by the TPF system when an IP address changes from active to inactive state. The linkdown trap includes the interface index from the interface table that has become inactive. This trap is sent for the following conditions:

- For CDLC IP addresses, when the last IP router associated with this IP address becomes inactive
- For OSA real IP addresses, when the OSA connection for the real IP address becomes inactive
- For VIPAs, when the OSA-Express connection that is currently associated with the VIPA becomes inactive and the VIPA is not swung to the alternate OSA-Express connection, or when a VIPA currently associated with an active OSA-Express connection is moved to another processor.

See “Operator Procedures for TCP/IP Native Stack Support” on page 59 for more information about CDLC IP addresses and OSA-Express and VIPA support.

### Linkup trap

Generated by the TPF system when an IP address changes its state from inactive to active. The linkup trap includes the interface index from the interface table that has become active. This trap is sent for the following conditions:

- For channel data link control (CDLC) IP addresses, when the first IP router associated with the IP address becomes active
- For Open Systems Adapter (OSA)-Express connection real IP addresses, when the OSA-Express connection for the real IP address becomes active
- For virtual IP addresses (VIPAs), when the OSA-Express connection currently associated with the VIPA becomes active, or when the VIPA is moved to an active OSA-Express connection on another processor or swung from an inactive OSA-Express connection to an active OSA-Express connection.

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 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 */

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);
}

```

---

## Installing TPF SNMP Agent Support

To install the SNMP agent server, do the following:

1. Install and define TPF TCP/IP native stack support.
2. Install the TPF SNMP agent.
3. Create the `/etc/snmp.cfg` SNMP configuration file and enter the ZSNMP command with the REFRESH parameter specified to refresh the configuration file and copy it into core storage.
4. Code the UCOM user exit and the UMIB user exit, if needed.
5. Define and start the SNMP agent server to the Internet daemon.
6. Define IP routing table entries, if needed to define alternate paths between TPF and the SNMP managers.
7. Define the TPF system to SNMP managers.

## Installing and Defining TPF TCP/IP Native Stack Support

TPF SNMP agent support requires TCP/IP native stack support. Ensure that TCP/IP native stack support is defined on your TPF 4.1 system before attempting to use SNMP agent support. See “TCP/IP Native Stack Support Internals” on page 39 and “Operator Procedures for TCP/IP Native Stack Support” on page 59 for more information about TCP/IP native stack support.

## Installing the SNMP Agent

Install SNMP agent support.

## Creating the SNMP Configuration File

The TPF system needs specific information to run correctly as an SNMP agent, including the values needed for the MIB. For example, there is a MIB value for the system name (`sysName`) that is supplied in the `/etc/snmp.cfg` SNMP configuration file. Specific keywords for the values are coded, followed by the values. Most of the information in the configuration file is not processor-unique information. However, the system name (`sysName`) and system object identifier (`sysObjectID`) are processor-unique. Because the file resides on the TPF file system that is shared by all processors, the TPF system needs a way to distinguish this information between processors. The TPF system does this by appending the processor ID to the

keyword for the value. So, if there are three processors in the complex, there can be three statements defining the system names for each processor.

To create the `/etc/snmp.cfg` SNMP configuration file, do the following:

1. Create the `snmp.cfg` SNMP configuration file.
2. Using the File Transfer Protocol (FTP), transfer the file to the `/etc` directory of the basic subsystem (BSS) file system.
3. From the BSS, enter the ZSNMP command with the REFRESH parameter specified to refresh the configuration file and copy it into core storage.

The following shows an example of an SNMP configuration file.

```
* SNMP SAMPLE CONFIG FILE *

SysDescription: "TPF Test System"
SysContact:     "TPF Administrator"
SysLocation:    "Poughkeepsie, NY"
*
SysNameA:       "TPF A"
SysNameB:       "TPF B"
SysObjIDA:      1.3.6.1.4.1.1.1
SysObjIDB:      1.3.6.1.4.1.1.2
*
CommName:       "public"
*
TrapIP: 19.17.153.14      * MANAGER 1 *
TrapIP: SNMP.MANAGER.COM * MANAGER 2 *
TrapIP: 2.26.7.114       * MANAGER 3 *
```

The SNMP configuration file contains keywords for identifying and describing your SNMP agent. Each keyword must be on its own line and must start at the first column of that line. Lines that do not start with a valid keyword are treated as comments. The keywords are not case-sensitive. The value of a keyword cannot exceed 255 characters in length. The keywords and their descriptions follow:

**SYSDescription:**

A textual description of the TCP/IP network node (in this case, the TPF system) that contains the full name and version identification of the type of system hardware, the software operating system, and the networking software. This keyword value must be in quotes.

**SysContact:**

A textual description of the person who manages this TCP/IP network node, as well as information about how to contact this person. This keyword value must be in quotes.

**SysLocation:**

The physical location of this TCP/IP network node. This keyword value must be in quotes.

**SysName:**

An administratively assigned name for this managed TCP/IP network node. By convention, this is the fully-qualified domain name of the network node. Each processor in a loosely coupled system can have its own SYSNAME keyword. The CPU ID is appended to the end of SYSNAME. This keyword value must be in quotes.

**SysObjID:**

The enterprise-specific authoritative description of the network management

subsystem contained in the TCP/IP network node. Each processor in a loosely coupled system may have its own SYSOBJID keyword. The processor ID is appended to the end of SYSOBJID.

**COMMNAME:**

The name of the community where the TPF SNMP agent resides. This keyword value must be in quotes.

**TRAPIP:**

The IP address or host name of the SNMP managers. This keyword is used to determine where to send traps. If the TRAPIP keyword is coded as NONE, no traps will be sent by the TPF system. You can code as many as 10 TRAPIP keywords. You can only specify one IP address or one host name per keyword.

The information in the SNMP configuration file is read into core storage during system cycle-up or when you enter the ZSNMP command with the REFRESH parameter specified. SNMP agent support does not work unless the SNMP configuration file is loaded into core storage. See *TPF Operations* for more information about the ZSNMP command.

## Coding the UCOM and UMIB User Exits

For an SNMP request to be accepted and processed by the TPF system, the SNMP manager must be verified. You must decide which SNMP managers are allowed to submit requests to your TPF system. The UCOM user exit validates the SNMP manager. You must add the necessary code to the UCOM user exit to determine which SNMP manager is allowed to view information in the TPF MIB database. The default system action is to reject all SNMP requests if the UCOM user exit is not coded.

The UMIB user exit is required for users who want to provide their own enterprise-specific MIB variables. For this, logic has to be added to the UMIB user exit to retrieve and then BER-encode enterprise-specific MIB variables. Input to the UMIB user exit is the unique object identifier of the variable being queried. The UMIB user exit returns the encoded object identifier of the variable returned, followed by the encoded value. Use the `tpf_snmp_BER_encode C/C++` function to do this. See the *TPF C/C++ Language Support User's Guide* for more information about the `tpf_snmp_BER_encode C/C++` function. See *ISO 8825 Part 1: Basic Encoding Rules* for more information about BER encoding. Go to <http://www.iso.ch/> to view ISO 8825.

For example, if object ID 1.3.6.1.4.1.20 is passed to the UMIB user exit and the value of the user MIB variable is found to be an integer of 20, the following must be returned:

06062B0601040114	020114
Encoded Object ID	Encoded Value

The default coding for the UMIB user exit is to not acknowledge any MIB variable it is asked about. See *TPF System Installation Support Reference* for more information about the UCOM and UMIB user exits.

## Defining and Starting the SNMP Agent Server

The SNMP agent must be defined and started by the Internet daemon. The Internet daemon monitors the network for packets destined for the SNMP agent server on well-known port 161. Define the SNMP agent server to the Internet daemon by

entering the ZINET command. To define the SNMP agent server to the Internet daemon and then start it, do the following:

1. From the BSS, define the SNMP agent server by entering:  
**ZINET ADD SERVER-SNMP PGM-CNMA PROTOCOL-UDP MODEL-WAIT PORT-161**
2. From the BSS, start the SNMP agent server by entering:  
**ZINET START SERVER-SNMP**

See *TPF Operations* for more information about the ZINET commands.

## Defining IP Routing Table Entries

An important consideration when installing your TPF SNMP agent support is to define it in a way that eliminates single points of failure in the network between TPF and the SNMP managers. Do this by using VIPAs or by defining multiple routers to CDLC IP addresses. However, you can set up multiple local IP interfaces to send the trap message across, by setting up routing table entries for the remote manager IP address. You can also use routing table entries when multiple SNMP managers reside on different networks. Routing table entries can be defined to reach managers on different networks by sending out the trap message to different interfaces in the TPF system.

To define routing table entries, enter **ZTRTE ADD**. See “Operator Procedures for TCP/IP Native Stack Support” on page 59 for more information about defining VIPAs and routing table entries and see *TPF Operations* for more information about the ZTRTE command.

## Defining the TPF System to the SNMP Manager

To function as an SNMP agent, the TPF system must be part of an SNMP-managed network. The remote SNMP manager must be configured with the TPF system IP address and community name. To configure the SNMP manager with the IP address and community name, review your SNMP manager product information or see your system administrator.



---

## Domain Name System Support

The Domain Name System (DNS) enables a client to dynamically determine the IP address of a server by providing the host name of the server as input. The DNS also allows you to pass an IP address as input to get the host name associated with that IP address as output.

DNS support on the TPF system contains both client and server functions.

---

### DNS Server

A TPF complex can have many different host names. For example, you can assign a different host name to each TPF application. Each of these TPF host names can be associated with one or more local IP addresses. DNS support enables the TPF system to be a DNS server for the host names that reside in your TPF complex.

When a remote client wants to start a connection with a TPF server application across a TCP/IP network, the client needs to get the IP address of the server. Typically, this is done by the client application issuing the `gethostbyname` function call, passing the host name of the server application as input. The request will be sent to the local DNS server of the client, which in turn will communicate with remote DNS servers to obtain the requested information. The request will eventually flow into the TPF system and be processed by the TPF DNS server code. This code will look up the requested host name in the TPF host name file, select one of the available IP addresses assigned to that host name (which also determines which TPF host will get this connection), and send the response to the DNS query back to the client.

In the DNS hierarchy, you must define the TPF system as the authoritative DNS server for all the host names in your TPF complex. This way, requests for information about TPF host names will be sent to the TPF system. Responses sent by the TPF system always indicate that the information should not be cached. This forces subsequent requests for the same host name to flow into TPF as well. This enables load balancing of TCP/IP connections to be done by the TPF system because every request for a connection to TPF flows into TPF. If information is cached by remote DNS servers or clients, the information can become out of date (for example, if an IP address fails). Another problem with allowing remote nodes to cache information is that load balancing would be very difficult to control because the decision for which TPF host a socket is assigned to is not centralized in one place.

The TPF DNS server will only respond to queries for host names that reside in the TPF system. The TPF DNS server is designed to load balance TCP/IP connections in your TPF complex. It cannot be used to manage host names external to the TPF system.

### TPF Host Name Table

The TPF host name table is created from information you define in the `/etc/host.txt` file that must be created in the basic subsystem (BSS). Each line in the file contains a TPF host name followed by zero or more local IP addresses that can be used for connections with this host name. If you do not specify any local IP addresses, all local IP addresses are candidates for use for connections with that host name. The same local IP address can be assigned to multiple host name entries. Blank characters are the only delimiter characters allowed in the file

between a host name and an IP address, or between IP addresses. Host names in the file can contain a wildcard character, which you can use only at the beginning of the host name. For example, \*.tpf.com is a valid host name but tpf.\*.com is not. You can also specify a host name that contains only a wildcard character. This entry will meet the criteria for any host name that is received, so you must place the entry last in the file; otherwise, all host name entries after the entry with the wildcard character will never be selected. Host names in the file can be lowercase or uppercase characters, but when the file is copied into the TPF host name table (THNT), they are converted to all uppercase characters.

The following shows an example of a TPF host name file:

```
Buytickets.com    1.1.1.2    1.1.2.6    1.1.4.9
Overcharge.org
US.SPEEDTRAPS.COM 111.111.111.111
NYR1940.org       1.1.2.6    2.2.2.222
res0.tpf.com      1.1.1.1
*.tpf.com         3.3.3.3    4.4.4.4
```

The following shows an example of a TPF host name file that is coded incorrectly. In this example, host name res0.tpf.com will never be selected because host name \*.tpf.com will be found first. The first matching host name in the TPF host name table will be the one selected even if there is a more exact match later in the table.

```
Buytickets.com    1.1.1.2    1.1.2.6    1.1.4.9
Overcharge.org
*.tpf.com         3.3.3.3    4.4.4.4
US.SPEEDTRAPS.COM 111.111.111.111
NYR1940.org       1.1.2.6    2.2.2.222
res0.tpf.com      1.1.1.1
```

The TPF host name table is created from the /etc/host.txt file. If you update the /etc/host.txt file, the TPF host name table is automatically refreshed in 1 minute after the update is made. You do not need to stop and restart the TPF DNS server to use information in the updated file.

## IP Address Selection

When a DNS request is received, the TPF host name table is examined to find the host name entry that matches the DNS request. If no match is found, the TPF system will reject the DNS request.

Once the TPF host name table entry is located, the DNS server code will construct a list of local IP addresses that are candidates for the answer to this DNS request. Initially, the list contains all the IP addresses listed in the TPF host name table entry. Next, the list is compacted so that only active IP addresses remain. For example, the TPF host name table entry could indicate that IP addresses X, Y, and Z are candidates; however IP address Y resides on a TPF processor that is not active. Only IP addresses X and Z then remain in the list.

After the final list has been constructed, the action to take is based on how many IP addresses are in the list:

- If there are no IP addresses in the list and the host name does not exist, the TPF system will reject the DNS request. If the host name does exist on the TPF system, but there are no active IP addresses to use, the TPF system sends back a positive DNS response indicating the host name was located, but the response does not contain the IP address to use.
- If there are two or more IP addresses in the list, the DNS select IP address user exit (UDNS) is called to select the IP address to use.

If the UDNS user exit is called, the input includes the TPF host name and the list of local IP addresses that can be used. Each IP address in the list also includes the CPU on which that IP address is active. The UDNS user exit does one of the following:

- Selects an IP address from the list
- Selects a CPU, where the TPF system will select an IP address from the list that resides on the specified CPU
- If there are no active IP addresses in the list for the selected CPU, the TPF system will send a positive DNS response, but not include any IP address in that response.
- Instructs the TPF system to select the IP address from the list.

If the TPF system selects the IP address, a round-robin algorithm is used.

---

## DNS Client

When a client application in the TPF system wants to start a connection with a remote server across a TCP/IP network and the client does not know the IP address of the server, it issues the `gethostbyname` function call. Before DNS support for PUT 13, the TPF system always sent a request to an external DNS server to obtain the IP address. DNS support enhances the DNS client function of the TPF system by providing a caching function. Now, whenever a `gethostbyname` function call is issued and information is received from an external DNS server, the information is saved in TPF DNS host name cache `IDNSHOSTNAME`. If subsequent `gethostbyname` calls are made for the same host name and the information is still in the cache, control is returned to the client application immediately and no flows to external DNS servers are needed. If you have many TPF client applications that constantly start connections with the same remote server applications, the DNS host name cache will improve the performance of those applications.

When the TPF system receives a DNS response from an external DNS server, the response indicates how long the information is valid. That value determines how long the information remains in the cache. After that time limit ends, the next `gethostbyname` request from that host name will cause another flow to the external DNS server.

If a DNS response from an external DNS server contains more than one IP address, all the IP addresses in the response are saved in the cache entry. Each time the cache entry is accessed, the order of the IP addresses passed back to the client application changes. For example, assume the cache entry for a given host name contains IP addresses X, Y, and Z. The first TPF client that issues a `gethostbyname` request for that host name will get back a list containing X, Y, and then Z. The next client to request this host name will get back a list containing Y, Z, and then X. The client after that will get back Z, X, and then Y. The list of IP addresses changes for load balancing purposes because it is common practice in applications to select the first IP address in the list.

In addition to the DNS host name cache, there is also a DNS IP address cache (`IDNSHOSTADDR`) that is used by the `gethostbyname` function call. For this, the client passes an IP address as input and requests the corresponding host name as output.

The quantity of entries in the DNS caches can be changed by using the ZCACH command. The name of the DNS host name cache is IDNSHOSTNAME. The name of the DNS IP address cache is IDNSHOSTADDR. See *TPF Operations* for more information about the ZCACH command.

If the TPF system is unable to obtain the `gethostbyname` or `gethostbyaddr` function information from the DNS local caches or the external DNS server, the system will attempt to obtain the information from the core copy of the optional `/etc/hosts` file if one has been set up. There can be only one IP address per host name entry and one entry per line in the `/etc/hosts` file. The `/etc/hosts` file must be created in the basic subsystem (BSS). Host names in the file can be lowercase or uppercase characters, but when the file is copied into the TPF host name table (THNT) they are converted to all uppercase. A sample `/etc/hosts` file is as follows:

```
Buytickets.com      1.1.1.2
Overcharge.org      1.2.3.4
US.SPEEDTRAPS.COM  111.111.111.111
NYR1940.org         1.1.2.6
```

---

## Internet Security

With growing TCP/IP networks, Internet security has become an important issue. There are two well-known types of security for the Internet:

- Secure Sockets Layer (SSL) allows applications to communicate in a secure manner over a TCP/IP network. Go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm> and click **SSL for the TPF 4.1 System: An Online User's Guide** for more information.
- Firewall support, which includes the following functions:
  - Protection against denial-of-service attacks
  - Proxies
  - Packet filtering.

The following information focuses on denial-of-service attacks and packet filtering. Proxies are application-specific and do not reside or belong on an application server (such as the TPF system).

---

## Denial-of-Service Attacks

Denial-of-service attacks are aimed at exposing weaknesses in an effort to take down an entire network or a particular server. The attacks that are of primary concern for the TPF system are those that attempt to take down a particular server. These attacks are an attempt to cripple a server by sending the server a packet or series of packets that will cause the server to crash or run out of memory or buffers. Some attacks are directed at specific applications, while others are aimed at the TCP/IP stack itself. Common denial-of-service attacks include:

- *The PING of death*; that is, sending a very large Internet Control Message Protocol (ICMP) echo message
- Sending IP fragments with overlapping sequence numbers
- *SYN attack* or *SYN flood*; that is, sending many TCP connection requests, but never completing the handshake.

The TPF system provides protection against known denial-of-service attacks directed toward TCP/IP stacks, along with safeguards to prevent many potential denial-of-service attacks in the future.

---

## Packet Filtering

The concept behind packet filtering is to examine each packet for an approved source and destination (that is, application). Packet filtering can be done in routers, but there are known ways to bypass packet filtering in routers by using fragmentation. The most secure implementation is to implement distributed packet filtering in both routers and hosts.

TCP/IP packet filtering firewall support allows you to define rules to filter inbound packets destined for TPF applications. The packets are filtered based on the source Internet Protocol (IP) address of the packet, the destination port of the packet, the protocol of the packet, and the action to take if the packet fits the rule.

The packet filtering rules are defined in a file called `/etc/iprules.txt`. To set up or modify the packet filtering rules, do the following:

1. Create or modify the `/etc/iprules.txt` file by doing one of the following:

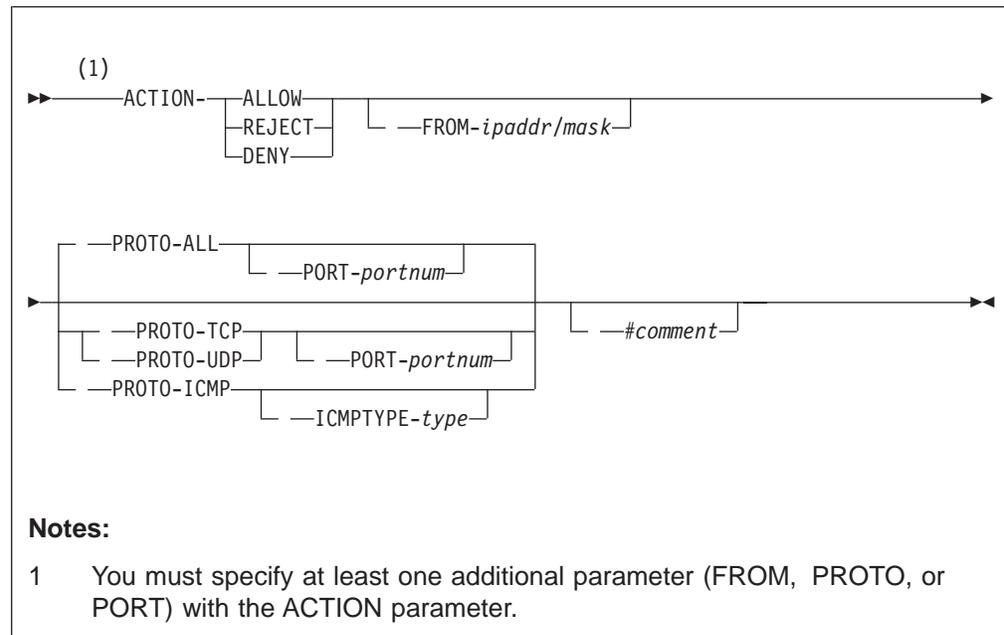
- Use the ZFILE commands to create or update the file directly on your TPF system.
  - Create or modify the file on another system and use Trivial File Transfer Protocol (TFTP) or File Transfer Protocol (FTP) to transfer the file to the basic subsystem (BSS) of your TPF system.
2. From the BSS, enter **ZFILT REFRESH** to refresh the file and copy it to core storage. The rules take effect immediately after you enter this command.

**Note:** Information from the packet filtering rules file is also read into core storage during system restart.

You can display the packet filtering rules that are defined in the TPF system by entering **ZFILT DISPLAY**. The display shows what rules are defined, as well as the number of packets that have applied to that rule. See *TPF Operations* for information about the ZFILE and ZFILT commands.

## Packet Filtering Rules File Syntax

Each line of the `/etc/iprules.txt` file can have a maximum of 300 characters and has the following syntax:



### ACTION

specifies the action to take if the packet matches the rule. Specify one of the following:

#### ALLOW

allows the packet to be processed by the TPF system.

#### DENY

discards the packet and takes no further action.

#### REJECT

discards the packet and responds to the remote client with a negative response. For TCP packets, the TPF system sends a reset (RST) message. For UDP or RAW packets, the TPF system sends an ICMP destination unreachable message.

**Note:** Certain ICMP messages defined by the ICMP architecture are not allowed to be rejected. The TPF system discards these messages.

**FROM-*ipaddr/mask***

specifies the IP network for the source of the packet, where:

*ipaddr*

is an IP address of the remote network in dotted decimal format.

*mask*

is a number, from 1 to 32, that represents the number of bits in *ipaddr* that represent the network portion of the address.

If you do not specify the FROM parameter, the rule applies to all IP addresses.

**PROTO**

specifies the protocol of the packet. Specify one of the following:

**TCP**

specifies the Transmission Control Protocol (TCP).

**UDP**

specifies the User Datagram Protocol (UDP).

**ICMP**

specifies the Internet Control Message Protocol (ICMP).

**ALL**

specifies all protocols.

**PORT-*portnum***

specifies the destination port of the packet, which is the port of the TPF application, where *portnum* is a decimal port number from 1 to 65535. If you do not specify this parameter, the rule applies to all port numbers.

**ICMPTYPE-*type***

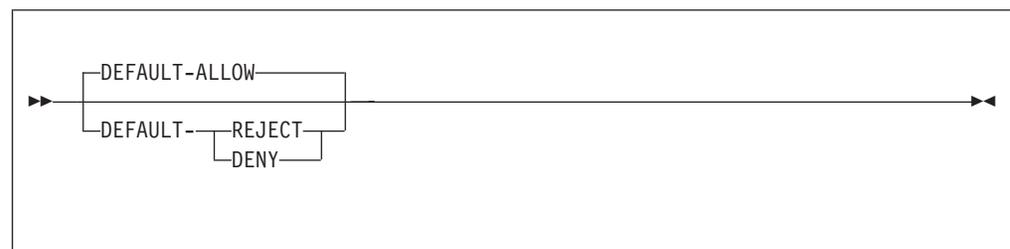
specifies the type of ICMP message, where *type* is a decimal number from 1 to 255. For example, ICMPTYPE-8 is an echo request (or PING message).

**#*comment***

is a comment associated with this entry. You can also code a comment on a separate input line. In general, blank lines and lines beginning with a # symbol are ignored.

**Packet Filtering Default Rule**

The last line of the `/etc/iprules.txt` file can be the default rule for the TPF system. The default rule is applied if no other rules apply to the packet. The default rule has the following syntax:



**DEFAULT**

specifies the action to take for any packets that do not match the other packet filtering rules. Specify one of the following:

### **ALLOW**

allows the packet to be processed by the TPF system.

### **DENY**

discards the packet and takes no further action.

### **REJECT**

discards the packet and responds to the remote client with a negative response. For TCP packets, the TPF system sends a reset (RST) message. For UDP or RAW packets, the TPF system sends an ICMP destination unreachable message.

**Note:** Certain ICMP messages defined by the ICMP architecture are not allowed to be rejected. The TPF system discards these messages.

If you do not code a default rule, the default action is set to ALLOW.

## **Considerations for Packet Filtering Rules**

You can define a maximum of 120 rules in the `/etc/iprules.txt` file. Code the rules that are most likely to be used at the start of the file. Keep the following in mind as you define the packet filtering rules for your TPF system:

- The order of the rules in the file will determine whether you get the results you expect.
- The number of rules defined in the file can affect performance if the majority of traffic is not TCP.

### **Order of Rules**

The order of the rules in the `/etc/iprules.txt` file is very important. Consider the following example:

```
ACTION-ALLOW PORT-1414
ACTION-REJECT FROM-9.117.249.0/24
```

In this example, all packets from network 9.117.249.0 will be rejected **except** those whose destination is port 1414. If the rules are reversed, **all** packets from network 9.117.249.0 will be rejected, **including** those whose destination is port 1414.

### **Performance Considerations**

To reduce packet filtering overhead, most TCP messages bypass packet filtering. Input messages received for existing TCP connections bypass packet filtering because the remote user has already been approved by packet filtering code when the connection was established. If a TCP input message other than a new connection request (that is, if it is not a SYN message) is received and the connection does not exist, packet filtering rules will be examined to determine if the packet should be rejected or discarded.

For non-TCP traffic, the more rules that you define, the more overhead there will be for packet filtering processing. The packet filtering rules are scanned for every UDP or RAW input packet.

## **Examples of Packet Filtering Rules**

The following are various examples of packet filtering rules.

- The following example rejects all packets whose destination is for port number 21 and received from the 9.117.249.0 network:

```
ACTION-REJECT FROM-9.117.249.0/24 PORT-21
```

- The following example allows a specific user with an IP address of 9.117.249.23 to access the TPF application at port 5600 using the UDP protocol:

```
ACTION-ALLOW FROM-9.117.249.23/32 PORT-5600 PROTO-UDP
```

- The following example discards packets from all users in network 9.121.0.0:

```
ACTION-DENY FROM-9.121.0.0/16
```

- The following example allows PING requests from users in network 9.117.249.0 and discards PING requests from users in other networks:

```
ACTION-ALLOW FROM-9.117.249.0/24 PROTO-ICMP ICMPTYPE-8
ACTION-DENY PROTO-ICMP ICMPTYPE-8
```

- The following example:

- Allows all packets whose destination is port 1414
- Allows all packets from the 9.117.249.0 network
- Denies all ICMP packets that are type 8, except those from the 9.117.249.0 network
- Rejects all other packets.

```
ACTION-ALLOW PORT-1414
ACTION-ALLOW FROM-9.117.249.0/24
ACTION-DENY PROTOCOL-ICMP ICMPTYPE-8
DEFAULT-REJECT
```

## Problem Diagnosis

You can use the IP trace facility to identify packets that violate the packet filtering rules. If an exception condition is associated with a packet, a reason code is added to the entry in the IP trace table. You can use the IP trace facility to search for packets with specific reason codes. Consider the following example.

Assume you entered ZFILT DISPLAY and received the following:

```
FILT0001I 11.05.14 DISPLAY PACKET FILTERING RULES
```

RULE	ACTION	REMOTE NETWORK	PORT	PROTO	ICMPTYPE	PACKETS
1	REJECT	9.117.249.0/24				224
2	ALLOW		21	TCP		5087
3	ALLOW			ICMP		87
4	ALLOW		520			21
DEF	DENY					14

```
END OF DISPLAY+
```

This display indicates that a number of packets from network 9.117.249.0 have been passed to the TPF system and rejected. You can use the offline IP trace facility to determine the IP addresses of the remote nodes that violated this rule and the TPF applications that the nodes were attempting to reach.

To find all the sent packets that have been rejected by the firewall, code an RC value of 01 on the PARM parameter of the IPTPRT JCL. For example:

```
PARM="RC 01"
```

The statement specifies that all packets with a reason code of REJECTED BY FIREWALL will be included in the IPTPRT report.

Figure 21 on page 110 shows an example of the resulting IPTPRT report:

```

*****
TRANSACTION PROCESSING FACILITY TCP/IP TRACE OUTPUT
*****
RECORDS MATCHING THE FOLLOWING SELECTION CRITERIA WILL BE PRINTED:
PROTOCOLS: . . . . . ALL
SOURCE PORTS: . . . . . ALL
DESTINATION PORTS: . . . . ALL
SOURCE IP ADDRESSES: . . . ALL
DESTINATION IP ADDRESSES: ALL
REASON CODES: . . . . . 01
IP CCW: . . . . . ALL
DATE: . . . . . FROM JAN01 TO DEC31
TIME: . . . . . FROM 00:00:00 TO 23:59:59
TOD (FIRST WORD): . . . . FROM 00000000 TO FFFFFFFF
TCP FLAGS: . . . . . ALL
WIDE LAYOUT
IP FORMATTED TRACE
RWI-02 IPCCW-D1 SOURCE IP-9.117.249.52 DEST IP-9.117.241.12 LEN-48
TOD-B6FA5951E20BF04E PROTOCOL-06 (TCP) SOURCE PORT-1029 DEST PORT-1414
SEQ-2491461275 WINDOW-65535 URGENT OFFSET-0
TCP FLAG BYTE-02 (SYN)
REASON CODE - REJECTED BY FIREWALL
IP HEADER 45000030 A70E0000 3906DD8E 0975F934 0975F10C
TCP HEADER 04050586 9480AE9B 00000000 7002FFFF 30FD0000 02040F00 01030304
RWI-01 IPCCW-D1 SOURCE IP-9.117.241.12 DEST IP-9.117.249.52 LEN-40
TOD-B6FA5951E6AC964F PROTOCOL-06 (TCP) SOURCE PORT-1414 DEST PORT-1029
SEQ-0 ACK-2802712577 WINDOW-0 URGENT OFFSET-0
TCP FLAG BYTE-14 (ACK, RST)
REASON CODE - REJECTED BY FIREWALL
IP HEADER 45000028 4B780000 3C06362D 0975F10C 0975F934
TCP HEADER 05860405 00000000 A70E0001 50140000 020B0000
RWI-02 IPCCW-D1 SOURCE IP-9.117.249.52 DEST IP-9.117.241.12 LEN-48
TOD-B6FA595AC1082B43 PROTOCOL-06 (TCP) SOURCE PORT-1030 DEST PORT-1414
SEQ-2511008644 WINDOW-65535 URGENT OFFSET-0
TCP FLAG BYTE-02 (SYN)
REASON CODE - REJECTED BY FIREWALL
IP HEADER 45000030 A70F0000 3906DD8D 0975F934 0975F10C
TCP HEADER 04060586 95AAF384 00000000 7002FFFF EAE80000 02040F00 01030304
RWI-01 IPCCW-D1 SOURCE IP-9.117.241.12 DEST IP-9.117.249.52 LEN-40
TOD-B6FA595AC489E66C PROTOCOL-06 (TCP) SOURCE PORT-1414 DEST PORT-1030
SEQ-0 ACK-2802778113 WINDOW-0 URGENT OFFSET-0
TCP FLAG BYTE-14 (ACK, RST)
REASON CODE - REJECTED BY FIREWALL
IP HEADER 45000028 4B7A0000 3C06362B 0975F10C 0975F934
TCP HEADER 05860406 00000000 A70F0001 50140000 02090000

:
:

```

Figure 21. IPTPRT Report Example

This report shows that the intruding IP address is 9.117.249.52, and that it is destined for the TPF server application with port 1414. With this information, you can take any appropriate action to resolve the problem.

See Appendix F, “Using the Internet Protocol Trace Facility” on page 407 for more information about the IP trace facility and for a complete description of the reason code values that you can specify.

**Note:** You can also use the ZIPTR or ZINIP command to display the IP trace information. See *TPF Operations* for more information about the ZIPTR and ZINIP commands and for examples of the displays.

---

## TCP/IP Network Services Database Support

The TCP/IP network services database contains information about TCP/IP server applications such as name, port, and protocol. This information is used by the `getservbyname` and `getservbyport` TCP/IP socket APIs, which allow you to retrieve the socket application port by passing the name and vice versa.

The TPF system supports an extended network services database. TCP/IP network services database support allows you to:

- Define quality of service (QoS) properties, including a differentiated services codepoint value for outbound messages.
- Define a weighting factor to allow messages for each application to be weighted differently when using data collection.

**Note:** The TCP/IP network services database is not a replacement for the Internet daemon. The application definitions in the TCP/IP network services database are independent from the definitions in the Internet daemon configuration file (IDCF); therefore, you need to define an application in both places if you want the Internet daemon to manage the application and if you want to also collect statistical data for the application. See “Operator Procedures for the Internet Daemon” on page 235 for more information about the Internet daemon and the IDCF.

---

## Quality of Service

The TCP/IP network services database uses quality of service (QoS) information to provide differentiated services for outbound messages, allowing the TPF system to mark outbound packets for each application to determine per hop behavior for routers.

The IP architecture, defined by Request for Comments (RFC) 791, originally defined a type of service (TOS) field in the IP header. This TOS field was redefined as a differentiated services field by RFC 2475. Each outbound TPF message has a differentiated services codepoint value that specifies the network priority. You can define this differentiated codepoint value for each application defined in the TCP/IP network services database by specifying the TOS parameter. You can also define a default value in keypoint 2 (CTK2) by using the IPTOS parameter on the SNAKEY macro. If you do not specify the TOS parameter for a particular application, the TPF system will use the default value defined in CTK2. If the outbound message is from a RAW socket, the TPF system will use the value of the IPTOS parameter unless it has already been filled in by the application. See *TPF ACF/SNA Network Generation* for more information about the SNAKEY macro.

See the following RFC documents for more information about the quality of service (QoS) architectures supported by the TPF system:

- RFC 2474 *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*
- RFC 2475 *An Architecture for Differentiated Services*.

Go to <http://www.ietf.org> for more information about these RFCs and any related extensions.

---

## Data Collection and Reduction

Information about the number of messages, bytes, and packets that are sent and received is collected for each TCP/IP application. This information is provided in the reports that are generated by data collection and reduction. These counters are updated and displayed differently based on whether or not the application is defined in the TCP/IP network services database, as well as how the application is defined in the database.

The byte and packet counts are updated by the TCP/IP stack. These counts are only updated for packets that contain data. Control messages such as SYN, SYN/ACK, and stand-alone ACKs are not counted.

If an application is defined in the TCP/IP network services database and has a weight parameter specified, the input and output message counts are updated by each individual TCP/IP application. See “Message Counts by Application” for more information about how to gather message counts on an application-by-application basis.

If an application is **not** defined in the TCP/IP network services database or does **not** have the weight parameter specified, the input and output message counts are updated as follows:

- The input message count is incremented each time a socket application issues a read, recv, recvfrom, activate\_on\_receipt, or activate\_on\_receipt\_with\_length function call and data is returned to the application.
- The output message count is incremented each time a socket application issues a write, writev, send, or sendto function call.

The number of TCP/IP weighted input messages is shown in the system summary report. In addition, there is a TCP/IP weighted input messages by application report and a TCP/IP message summary report. In these reports, the count information is shown for each individual application that is defined in the TCP/IP network services database. If an application is **not** defined in the database, the counts for that application are shown in a single category in the report called OTHER. See *TPF System Performance and Measurement Reference* for more information about data collection and reduction and for an example of the reports.

## Message Counts by Application

If you want to gather input and output message counts on an application-by-application basis, do the following:

1. Define each TCP/IP application for which you want message counts in the TCP/IP network services database and specify the weight parameter. See “TCP/IP Network Services Database File” on page 113 for more information about how to define applications for the database.

The weight parameter specifies a weighting factor for each message sent or received by a particular application. A value of 100 is equal to 1 message. Therefore, if you specify 50 for the weight parameter for an application, the messages for that application will count half as much. For example, if this application receives 50 messages per second, the input message count will indicate that the application received 25 weighted messages per second. Similarly, if you specify 200 for the weight parameter for an application, the messages for that application will count twice as much. For example, if this

application sends 50 messages per second, the output message count will indicate that the application sent 100 weighted messages per second.

2. Update your TCP/IP applications to call the `tpf_tcpip_message_cnt` function.

The TCP architecture does not have a single definition for a message; therefore, the definition of a message can be different for each application. For example, a single socket send can be considered to be one complete message, multiple messages, or part of a message. Use the `tpf_tcpip_message_cnt` function to increment the input or output message counters for your TCP/IP applications based on the definition of a message for that application. This function requires the port number and protocol of the application as input. If the application does not know the port number, the application can issue a call to the `getservbyname` socket API to get the port number. See the *TPF C/C++ Language Support User's Guide* for more information about the `tpf_tcpip_message_cnt` function.

The TCP/IP applications that are shipped with the TPF system (MQSeries, the TPF Internet mail servers, and so on) include calls to the `tpf_tcpip_message_cnt` function to increment the message counters. To count messages for these applications, you only need to define them in the TCP/IP network services database. See "TCP/IP Network Services Database File Example" on page 114 for an example of the definitions you can add for these applications and for information about how a message is defined for each of these applications.

---

## TCP/IP Network Services Database File

The TCP/IP network services database is created from information you define in a file called `/etc/services`. The `/etc/services` file contains entries for each application in the database. To set up or modify the TCP/IP network services database, do the following:

1. Create or modify the `/etc/services` file by doing one of the following:
  - Use the ZFILE commands to create or update the file directly on your TPF system.
  - Create or modify the file on another system and use Trivial File Transfer Protocol (TFTP) or File Transfer Protocol (FTP) to transfer the file to the basic subsystem (BSS) of your TPF system.
2. From the BSS, enter **ZIPDB REFRESH** to refresh the file and copy it to core storage. The definitions take effect immediately after you enter this command.

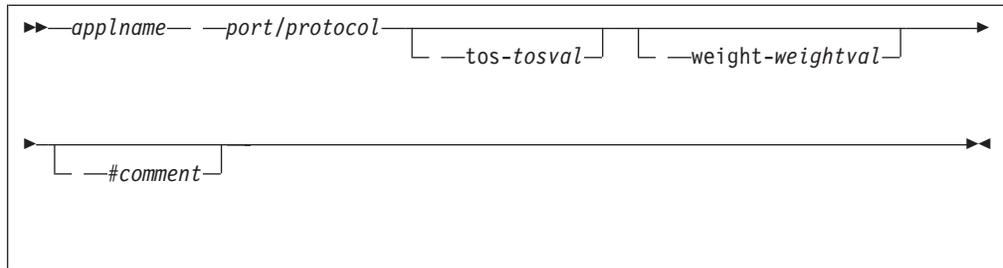
**Note:** Information from the TCP/IP network services database file is also read into core storage during system restart.

See *TPF Operations* for information about the ZFILE and ZIPDB commands.

To prevent port number conflicts with client sockets, define all TPF TCP/IP server applications that have a port number in the range 1024–5000 in the TCP/IP network services database.

## TCP/IP Network Services Database File Syntax

You can define a maximum of 1000 applications in the TCP/IP network services database file. Each line of the `/etc/services` file has the following syntax:



***applname***

is the 1- to 9-character name of the application to be defined in the TCP/IP network services database. Do not define an application with the name of OTHER; this name is reserved for IBM use.

***port***

is the port associated with the specified application.

***protocol***

is the protocol associated with the specified application. Specify one of the following:

**TCP**

specifies the Transmission Control Protocol (TCP).

**UDP**

specifies the User Datagram Protocol (UDP).

***tos-tosval***

specifies the differentiated services codepoint value to use for the network priority of outbound TPF IP packets for the specified application, where *tosval* is a value from 0 to 255. If you do not specify this parameter, the value defined by the IPTOS parameter on the SNAKEY macro is used.

***weight-weightval***

specifies the weighting factor to use when counting the messages for the specified application, where *weightval* is a value from 1 to 1000. A value of 100 equals one message. If you do not specify this parameter, messages will be counted as socket reads and writes. See "Data Collection and Reduction" on page 112 for more information about how TCP/IP messages are counted.

***#comment***

is a comment associated with this entry. You can also code a comment on a separate input line. In general, blank lines and lines beginning with a # symbol are ignored.

The following shows some sample entries:

```
rip          520/udp    weight-50      #RIP
smtp         25/tcp      #Simple Mail Transfer Protocol
mq           1414/tcp   tos-5 weight-100 #MQ Series
dns          53/udp     tos-10         #DNS
```

## TCP/IP Network Services Database File Example

Figure 22 on page 115 shows an example of a TPF /etc/services file with definitions for the TCP/IP server applications that are shipped with the TPF system. Table 6 on page 115 provides information about the definition of input and output messages for each of these applications.

ftp-data	20/tcp	weight-100	#FTP data
ftp-ctl	21/tcp	weight-100	#FTP control
smtp	25/tcp	weight-100	#Simple Mail Transfer Protocol
dns	53/udp	weight-100	#DNS
tftp	69/udp	weight-100	#Trivial File Transfer
http	80/tcp	weight-100	#World Wide Web
pop3	110/tcp	weight-100	#Post Office Protocol - Version 3
imap	143/tcp	weight-100	#Internet Message Access Protocol
snmp	161/udp	weight-100	#SNMP
snmp-trap	162/udp	weight-100	#SNMP trap
matipa	350/tcp	weight-100	#MATIP Type A
matipb	351/tcp	weight-100	#MATIP Type B
https	443/tcp	weight-100	#Secure HTTP
rip	520/udp	weight-100	#RIP
mq	1414/tcp	weight-100	#MQ
ipbridge	9500/tcp	weight-100	#IPBRIDGE
rpc	port1/tcp	weight-100	#RPC
tpfar	port2/tcp	weight-100	#TPFAR

Figure 22. /etc/services File Example. In this example, port1 and port2 represent the server ports that are defined for the RPC and TPFAR applications in your TPF system.

Table 6. Message Definitions for TPF TCP/IP Server Applications

Application	Input Message Definition	Output Message Definition
FTP-DATA	File transferred to the TPF system by using FTP	File sent from the TPF system by using FTP
FTP-CTL	FTP connection request received	FTP connection request accepted
SMTP	Mail message received	Mail message sent
DNS	DNS request received	Response to a DNS request sent
TFTP	File transferred to the TPF system by using TFTP	File sent from the TPF system by using TFTP
HTTP	Each HTTP request received	Response to an HTTP request sent
POP3	POP3 command received	POP3 command response sent
IMAP	IMAP command received	IMAP command response sent
SNMP	SNMP request received	SNMP response sent
SNMP-TRAP	N/A	SNMP trap sent to the SNMP manager
MATIPA	Each MATIP Type A message received	Each MATIP Type A message sent
MATIPB	Each MATIP Type B message received	Each MATIP Type B message sent
HTTPS	Each secure HTTP request received	Response to a secure HTTP request sent
RIP	RIP request received	RIP response or unsolicited RIP message sent
MQ	Each MQSeries message received	Each MQSeries message sent
IPBRIDGE	Each input message sent over the IP bridge	Each output message sent over the IP bridge
RPC	Each RPC message received from the client	Each RPC message sent to the client
TPFAR	SQL response received from DB2	SQL request that causes a flow to DB2



---

## Part 4. Socket Application Programming Interface Overview

<b>Socket Overview</b> . . . . .	119
Sockets . . . . .	119
Types of Sockets Supported by TCP/IP . . . . .	119
Socket Address for the Internet Domain . . . . .	119
Port Numbers . . . . .	120
Standard Dotted Decimal Formats . . . . .	120
Mapping Address Parts . . . . .	120
Integer Byte Order Conversion . . . . .	120
Blocking and Nonblocking . . . . .	121
Out-Of-Band Data . . . . .	121
TPF Socket Application Programming Interface (API) Support . . . . .	122
Socket API Functions Using TCP/IP Offload Support . . . . .	122
Socket API Functions Using TCP/IP Native Stack Support . . . . .	122
Socket User Exits . . . . .	122
Socket Accept for TCP/IP Offload Support . . . . .	123
Socket Activation. . . . .	123
Socket Connect . . . . .	123
Socket Cycle-Up When Using TCP/IP Offload Support . . . . .	123
Socket Deactivation. . . . .	123
Socket System Error . . . . .	124
TCP/IP Native Stack Support Accept Connection . . . . .	124
Select TCP/IP Support . . . . .	124
Socket Cycle-Up When Using TCP/IP Native Stack Support . . . . .	124
Full-Duplex Socket Support . . . . .	124
Using activate_on_receipt . . . . .	125
Socket Sweeper Support to Close Inactive Sockets . . . . .	125
<b>Sample Socket Sessions</b> . . . . .	127
Function Calls Used in a Sample TCP Session . . . . .	127
Using the activate_on_receipt Function Call . . . . .	129
Function Calls Used in a Sample UDP Session . . . . .	131
Main Socket Function Calls . . . . .	132
<b>Socket Application Programming Interface Functions Reference</b> . . . . .	137
General Function Information . . . . .	137
accept — Accept a Connection Request . . . . .	138
activate_on_accept — Activate a Program When the Client Connects . . . . .	141
activate_on_receipt — Activate a Program after Data Received . . . . .	144
activate_on_receipt_with_length — Activate a Program after Data of Specified Length Received . . . . .	148
bind — Bind a Local Name to the Socket. . . . .	152
close — Shut Down a Socket . . . . .	155
connect — Request a Connection to a Remote Host . . . . .	157
gethostbyaddr — Get Host Information for IP Address . . . . .	160
gethostbyname — Get IP Address Information by Host Name . . . . .	162
gethostid — Return Identifier of Current Host . . . . .	164
gethostname — Return Host Name . . . . .	166
getpeername — Return the Name of the Peer . . . . .	168
getservbyname — Get Server Port by Name . . . . .	170
getservbyport — Get Server Name by Port . . . . .	172
getsockname — Return the Name of the Local Socket. . . . .	174
getsockopt — Return Socket Options . . . . .	176
htonl — Translate a Long Integer. . . . .	180

htons — Translate a Short Integer . . . . .	181
inet_addr — Construct Internet Address from Character String . . . . .	182
inet_ntoa — Return Pointer to a String in Dotted Decimal Notation . . . . .	184
ioctl — Perform Special Operations on Socket . . . . .	185
listen — Complete Binding, Create Connection Request Queue . . . . .	189
ntohl — Translate a Long Integer. . . . .	191
ntohs — Translate a Short Integer . . . . .	192
read — Read Data on a Socket . . . . .	193
recv — Receive Data on a Connected Socket . . . . .	196
recvfrom — Receive Data on Connected/Unconnected Socket . . . . .	199
recvmsg — Receive Message on Connected/Unconnected Socket . . . . .	202
select — Monitor Read, Write, and Exception Status . . . . .	205
send — Send Data on a Connected Socket . . . . .	206
sendmsg — Send Message on a Socket . . . . .	210
sendto — Send Data on an Unconnected Socket. . . . .	212
setsockopt — Set Options Associated with a Socket. . . . .	216
shutdown — Shut Down All or Part of a Duplex Connection . . . . .	220
sock_errno — Return the Error Code Set by a Socket Call . . . . .	222
socket — Create an Endpoint for Communication. . . . .	223
write — Write Data on a Connected Socket . . . . .	226
writew — Write Data on a Connected Socket . . . . .	229

---

## Socket Overview

This chapter:

- Describes the types of data that the socket API functions handle
- Explains the characteristics of each socket type
- Describes commonly used socket programming concepts
- Discusses TPF socket application programming interface (API) support
- Discusses the socket user exits
- Discusses the TPF-unique `activate_on_receipt` and `activate_on_accept` socket functions
- Discusses socket sweeper support (closing inactive sockets).

---

## Sockets

Transmission Control Protocol/Internet Protocol (TCP/IP) support provides a set of C functions that applications can use to access the Internet for transferring or receiving data called *socket* APIs.

### Types of Sockets Supported by TCP/IP

Table 7 shows the types of sockets supported by TCP/IP and the related protocol generally associated with each protocol.

Table 7. Socket Types and Associated Data

Socket type	Protocol	Description
SOCK_STREAM	Transmission Control Protocol (TCP)	The stream socket (SOCK_STREAM) interface defines a reliable connection-oriented service. Data is sent without errors or duplication and is received in the same order as it is sent.
SOCK_DGRAM	User Datagram Protocol (UDP)	The datagram socket (SOCK_DGRAM) interface defines a connectionless service for datagrams, or messages. Datagrams are sent as independent packets. The reliability is not guaranteed, data can be lost or duplicated, and datagrams can arrive out of order. However, datagram sockets have improved performance capability over stream sockets and are easier to use.
SOCK_RAW	IP, ICMP, RAW	The raw socket (SOCK_RAW) interface allows direct access to lower-layer protocols such as Internet Protocol (IP).

**Note:**

The type of socket you use is determined by the data you are transmitting:

- When you are transmitting data where the integrity of the data is high priority, you **must** use stream sockets.
- When the data integrity is not high priority (for example, for terminal inquiries), use datagram sockets because of their ease of use and higher performance capability.

### Socket Address for the Internet Domain

A socket address for the internet domain is made up of 4 distinct parts defined by 16 bytes:

1. The first 2 bytes contain the domain parameter, which indicates the address space where communication is taking place.
2. The next 2 bytes contain the port number, which the TCP/IP software used to differentiate between different applications using the same protocol (TCP or UDP).

3. The next 4 bytes contain the internet address, which represents a unique network interface.
4. The remaining bytes in the 16-byte structure are not used.

The internet domain is the only address domain supported by TCP/IP support.

---

## Port Numbers

A *port* is an endpoint for communication between applications, generally referring to a logical connection. A port provides queues for sending and receiving data. Each port has a port number for identification.

- Port numbers 0 to 1023 are used for well-known ports.
- Port numbers 1024 to 65535 are available for the following user applications:
  - Port numbers 1024 to 5000 are reserved for clients.
  - Port numbers 5001 to 65535 are reserved for user server applications.

---

## Standard Dotted Decimal Formats

Values specified in standard dotted decimal notation take one of the following forms:

a.b.c.d  
a.b.c  
a.b  
a

## Mapping Address Parts

- When a four-part address is specified, each part is interpreted as a byte of data and assigned, from left to right, to one of the 4 bytes of an internet address.
- When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the 2 rightmost bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as 128.net.host.
- When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the 3 rightmost bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as net.host.
- When a one-part address is specified, the value is stored directly in the network address space without any rearrangement of its bytes.
- Numbers supplied as address parts in standard dotted decimal notation can be decimal, hexadecimal, or octal. Numbers are interpreted in C language syntax:
  - A leading 0x implies *hexadecimal*.
  - A leading 0 implies *octal*.
  - A number without a leading 0 implies *decimal*.

## Integer Byte Order Conversion

The following calls provide integer byte order conversions:

- `htonl` converts from host-to-network byte order operating on long unsigned\_integers.
- `htons` converts from host-to-network byte order operating on short unsigned\_integers.

- `ntohl` converts from network-to-host byte order operating on long `unsigned_integers`.
- `ntohs` converts from network-to-host byte order operating on short `unsigned_integers`.

**Notes:**

1. A long integer is 32 bits: for example, an internet address.
2. A short integer is 16 bits: for example, a port number.
3. The TPF system does not use the integer byte order conversion functions because the host byte order used in the TPF system is equivalent to the network byte order.

---

## Blocking and Nonblocking

Sockets can be set to either blocking or nonblocking I/O mode. The `FIONBIO` option on the `ioctl` call determines the mode. When `FIONBIO` is set, the socket is marked *nonblocking*. If a `read`, `recv`, or `recvfrom` function is tried and the desired data is not available, the socket does not wait for the data to become available but returns immediately with the `SOCWOULDBLOCK` secondary error code.

When `FIONBIO` is not set, the socket is in *blocking* mode. If a `read`, `recv`, or `recvfrom` function is tried and the desired data is not available, the calling process waits for the data. When a socket is created, the default mode is *blocking*.

Calls that are affected by the `FIONBIO` flag are:

- `accept`
- `connect`
- `read`
- `recv`
- `recvfrom`
- `recvmsg`
- `send`
- `sendmsg`
- `sendto`
- `write`
- `writv`.

---

## Out-Of-Band Data

The `MSG_OOB` flag is set using the `send` call on stream sockets. It indicates the presence of additional data being sent on a different channel and also indicates a change in the order in which data is read. For example, if you send a buffer of data, but want other data read first, you can send the data *out-of-band* (OOB).

Out-of-band data is always read before buffered data and, by using the OOB mark, you can indicate the exact sequence in which you want the data read. Therefore, the OOB mark allows the receiver to synchronize with the sender the order to read data in the normal data stream.

The `MSG_OOB` flag sends out-of-band data on sockets that support it. This option is only applicable to stream sockets. The sending side, using the `send` call, sends this data before any buffered data. Similarly, the receiving side, using the `recv` or `recvfrom` call, receives this data before any data that it might have buffered.

---

## TPF Socket Application Programming Interface (API) Support

Socket API support consists of:

- Socket API functions
- Socket user exits.

### Socket API Functions Using TCP/IP Offload Support

Socket API functions, which are issued by the socket application, enter socket API support to start the processing of the functions. If the socket is using offload support, socket API support either passes a return code back to the application or sends the function to the TCP/IP offload device to obtain the return code for the application.

If socket API support sends the function to the TCP/IP offload device, it builds a request in inter-user communication vehicle (IUCV) format and sends the function through the TPF CLAW device interface to the TCP/IP offload device, using Common Link Access to Workstation (CLAW) protocol. While it is waiting for the response from the TCP/IP offload device, socket API support issues EVNTC and EVNWC macros to suspend the ECB unless the function call was an `activate_on_receipt`. In the `activate_on_receipt` case, socket API support returns to the socket application after sending the function to the TCP/IP offload device.

If the function call issued to the TCP/IP offload device was not an `activate_on_receipt`, the return code from the TCP/IP offload device is passed to socket OPZERO by the TPF CLAW device interface. Socket OPZERO posts the suspended ECB to reactivate socket API support, which returns the return code back to the application. See “TCP/IP Internals” on page 11 for additional information about the TCP/IP internals and see “Socket Application Programming Interface Functions Reference” on page 137 for a complete description of the socket API calls.

### Socket API Functions Using TCP/IP Native Stack Support

Socket API functions, which are issued by the socket application, enter socket API support to start the processing of the functions. If the socket is using TCP/IP native stack support, the function is processed locally by the TPF system. If the operation can be completed immediately or the socket is running in nonblocking mode, control is returned to the application without a loss of control. If the operation cannot be completed immediately and the socket is running in blocking mode, the application ECB is suspended until the function is completed successfully, fails, or times out.

---

## Socket User Exits

Socket user exits allow you to activate and deactivate socket applications, and verify remote clients. TCP/IP offload support provides the following user exits:

- Socket accept for TCP/IP offload support
- Socket activation
- Socket connect
- Socket cycle-up
- Socket deactivation
- Socket system error

TCP/IP native stack support provides the following user exits:

- Socket cycle-up

- TCP/IP native stack support accept connection
- Select TCP/IP support.

See *TPF System Installation Support Reference* for additional information about the socket user exits.

## Socket Accept for TCP/IP Offload Support

The socket accept for TCP/IP offload support user exit (C542) provides a centralized program to screen all connection requests before they are returned to the server application. C542 is entered each time the `accept()` API function call receives a connection request from a client by using TCP/IP offload support.

## Socket Activation

The *socket activation* user exit is entered during system cycle-up and during ZCLAW ACTIVATE processing at or above CRAS state. This user exit activates socket server applications and is called for **each** TCP/IP offload device connected to the TPF system.

## Socket Connect

The *socket connect* user exit is called during CLAW connect processing and allows Common Link Access to Workstation (CLAW) applications to be activated.

## Socket Cycle-Up When Using TCP/IP Offload Support

The *socket cycle-up TCP/IP offload support* user exit is entered only once during system cycle-up after all active offload devices are connected to the TPF system. When the TPF system successfully issues the socket cycle-up user exit, this user exit activates server applications for **all** TCP/IP offload devices that are connected to the TPF system.

Following are the major differences between the socket activation user exit and the socket cycle-up user exit:

- When the exit is called.

The socket activation user exit is called **during** system cycle-up and **during** processing of the ZCLAW ACTIVATE Command at CRAS state or above. The socket cycle-up user exit is called only during system cycle-up.

- The number of offload devices affected.

The socket activation user exit is called for **each** offload device while the socket cycle-up user exit is called once to activate servers on **all** offload devices that are connected to the TPF system.

## Socket Deactivation

The *socket deactivation* user exit is entered:

- When the operator issues a ZCLAW INACTIVATE Command to deactivate a TCP/IP offload device logical link on an adapter
- During system cycle-down when all TCP/IP offload devices are disconnected
- When the CLAW device failure entry point (EP) is entered during the failure of a TCP/IP offload device
- When the CLAW DISCONNECT entry point (EP) is entered during the DISCONNECT of a TCP/IP offload device.

For each TCP/IP offload device disconnected from the TPF system, socket deactivation enters the socket deactivation user exit to deactivate socket applications associated with the TCP/IP offload device.

## Socket System Error

The *socket system error* user exit is entered when a socket application must exit as a result of a system error.

## TCP/IP Native Stack Support Accept Connection

The *TCP/IP native stack support accept connection (UACC)* user exit allows you to verify a remote client connection request. UACC is called when an `accept` or `activate_on_accept` request is made for a socket that uses TCP/IP native stack support.

## Select TCP/IP Support

The *select TCP/IP support (USOK)* user exit is called whenever a TPF application issues the socket call to create a new socket and both TCP/IP offload support and TCP/IP native stack support are defined. USOK decides whether the socket being created will use TCP/IP offload support or TCP/IP native stack support. The default will be to use TCP/IP offload support.

## Socket Cycle-Up When Using TCP/IP Native Stack Support

The *socket cycle-up (CLCV)* user exit is called once during system cycle-up and allows you to activate your socket server applications from a central point.

---

## Full-Duplex Socket Support

The TCP/IP system allows up to 16 different type calls to be outstanding for a particular socket descriptor. A *different type* call is a call that performs a different specific function. Following is a list of these different types of function calls:

- `accept` or `activate_on_accept` (for TCP/IP native stack support only)
- `read`, or `recv`, or `recvfrom`, or `activate_on_receipt`
- `send`, or `sendto`, or `write`, or `writetv`
- `connect`
- `shutdown`
- `close`
- `listen`
- `bind` or `socket`
- `gethostid`
- `gethostname`
- `getpeername`
- `getsockname`
- `getsockopt`
- `ioctl`
- `select`
- `setsockopt`.

Full-duplex socket support allows data to be sent and received on the same socket from different ECB programs without being blocked by the TPF system, the TCP/IP offload device, or the IP router.

If function calls from more than one ECB to the same socket are of the same type, blocking occurs. For example, if one ECB issues a read function call and another ECB issues a `recv` function call to the same socket, the second call is blocked until processing of the first function call is completed successfully.

---

## Using `activate_on_receipt`

When many applications issue blocked read, `recv`, or `recvfrom` calls, there could be many ECBs suspended at the same time if no data is available to be returned to the application. To reduce the number of ECBs suspended at the same time, a unique TPF socket API function called `activate_on_receipt` is available. This function allows an ECB to continue processing or to exit while the application waits for data to arrive from the TCP/IP network. When data arrives, another ECB is created to read the message. See “`activate_on_receipt` — Activate a Program after Data Received” on page 144 for more information about the `activate_on_receipt` API function.

---

## Socket Sweeper Support to Close Inactive Sockets

If communication across a socket stops, the socket will still stay open until one of the following occurs:

- The TCP/IP offload device associated with the socket is deactivated (only for offload sockets).
- System cycle-down below CRAS state.
- The socket is closed by the application.

Socket sweeper support closes inactive socket descriptors after a specified period of time so the inactive socket descriptors can be free for another communication session.

You can specify the period of time with the `SOCKSWP` parameter of the `SNAKEY` macro, and change the period of time online with the `ZNKEY` command using the `SOCKSWP` parameter.

The `SOCKSWP` parameter specifies, in minutes, the interval for the socket sweeper. Socket application programs that do not use a particular socket descriptor in the period of time that you specify will be closed by the socket sweeper program.

If you specify zero, the socket sweeper does not close any inactive socket descriptors. For this, the socket application code must close inactive sockets.

For sockets using TCP/IP native stack support, the socket sweeper will also close sockets if all IP routers that could be used by the socket are not active for an extended period of time.



---

## Sample Socket Sessions

This chapter:

- Shows the general sequence of calls for socket sessions using Transmission Control Protocol (TCP)
- Shows the general sequence of calls using the `activate_on_receipt` function call
- Shows the general sequence of calls for socket sessions using User Datagram Protocol (UDP)
- Discusses the input and output for the more commonly used socket API functions.

---

### Function Calls Used in a Sample TCP Session

In a TCP connected stream socket session, the roles of server and client are more clearly defined than in a datagram socket session. Once you make the connection, the connection exists until you close the socket. A connected socket, as used in TCP protocol, sends data to and receives data from only one server because it has a dedicated destination. The following steps correspond to the numbers in Figure 23 on page 128:

1. The server is started first by issuing a socket call to create socket **s**. The client then issues a socket call and creates its own socket **s**. A socket is initially created in the unconnected state, which means that the socket is not associated with any remote destination.
2. The server issues a `bind` call to a local address to be positioned for a subsequent connection.  
The client can issue an *optional* `bind` call to a local address.
3. Using the `listen` function, the server waits for a connection from the client.
4. The `connect` call places the socket in the connected state. The client **must** issue the `connect` call before being able to transfer data through a reliable stream socket.
5. The server issues an `accept` call to accept an incoming connection. To allow the server socket **s** to remain available for the next client connection, the `accept` call creates a new socket **ns**, which is dedicated to the client.
6. The `read` and `write` calls between the client and server continue until all the data is transferred.
7. The client closes socket **s** and the server closes the related socket, socket **ns**.
8. The server can continue to accept other connections on the original socket **s** or close it by using the `close` function.

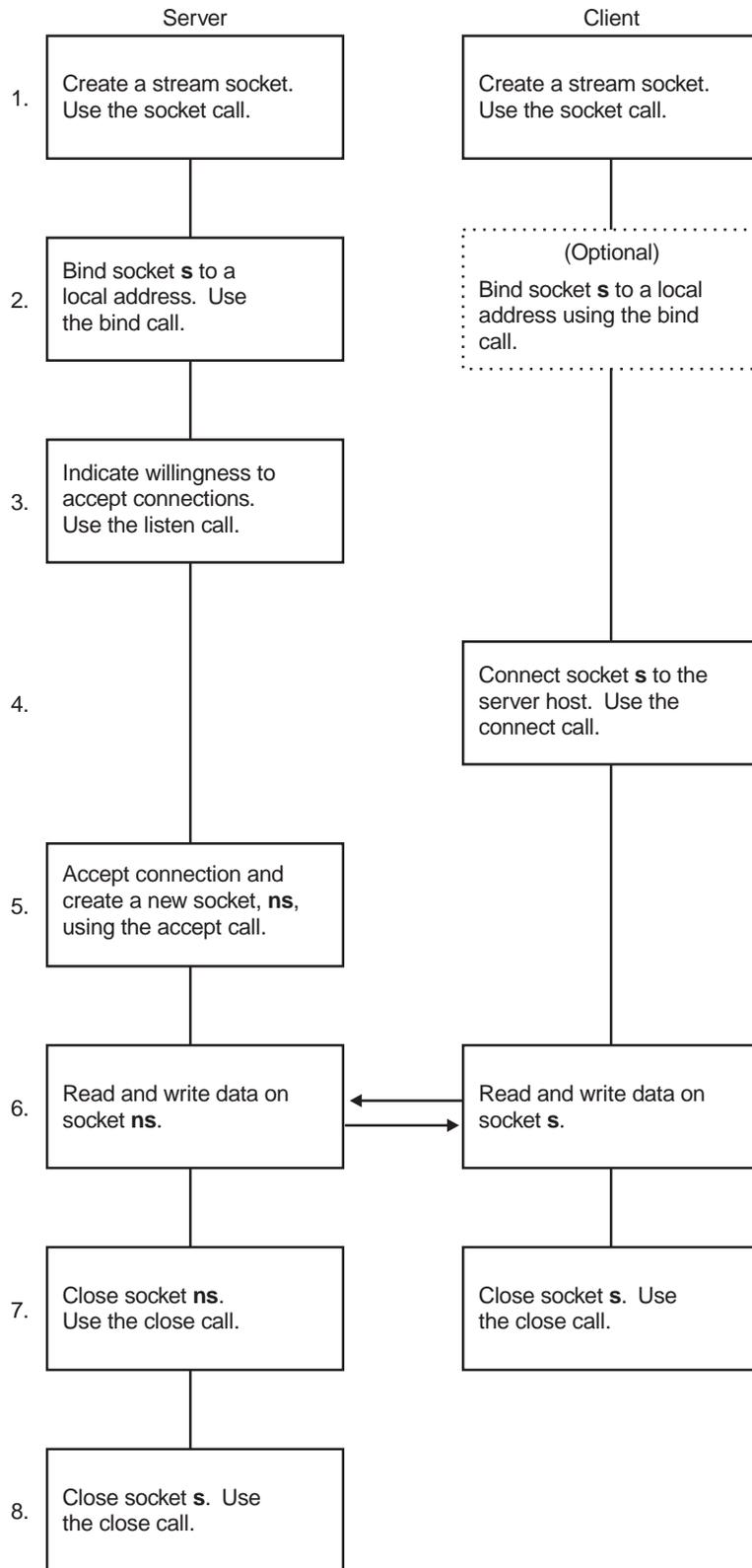


Figure 23. Sample Socket Session Using TCP Protocol

---

## Using the activate\_on\_receipt Function Call

The TPF system provides the `activate_on_receipt` function, a unique TPF socket API function designed to prevent many ECBs from being suspended at the same time. See the drivers in Appendix D, “Sample Application Driver Code” on page 367 for an example of how to use the `activate_on_receipt` function. The following steps correspond to the numbers in Figure 24 on page 130:

1. The server is started first by issuing a socket call to create socket **s**. The client then issues a socket to create its own socket **s**. A socket is initially created in the unconnected state, which means that the socket is not associated with any remote destination.
2. Using the `listen` function, the server issues a `bind` call to a local address to be positioned for a subsequent connection.  
The client can issue an *optional* `bind` call to a local address.
3. The server waits for a connection from the client.
4. The `connect` call places the socket in the connected state. The client **must** issue the `connect` call before being able to transfer data through a reliable stream socket.
5. The server issues an `accept` call to accept an incoming connection. To allow the server socket **s** to remain available for the next client connection, the `accept` call creates a new socket **ns**, which is dedicated to the client.
6. The server issues an `activate_on_receipt` function to create a new ECB and activate a new child server program when data is received from the client. The original server becomes a *parent server*.
7. When the child server program is activated, the `read` and `write` calls between the client and child server continue until all the data is transferred.
8. The client closes socket **s** and the child server closes the related socket, socket **ns**. The parent server can continue to accept other connections on the original socket **s** or close it.

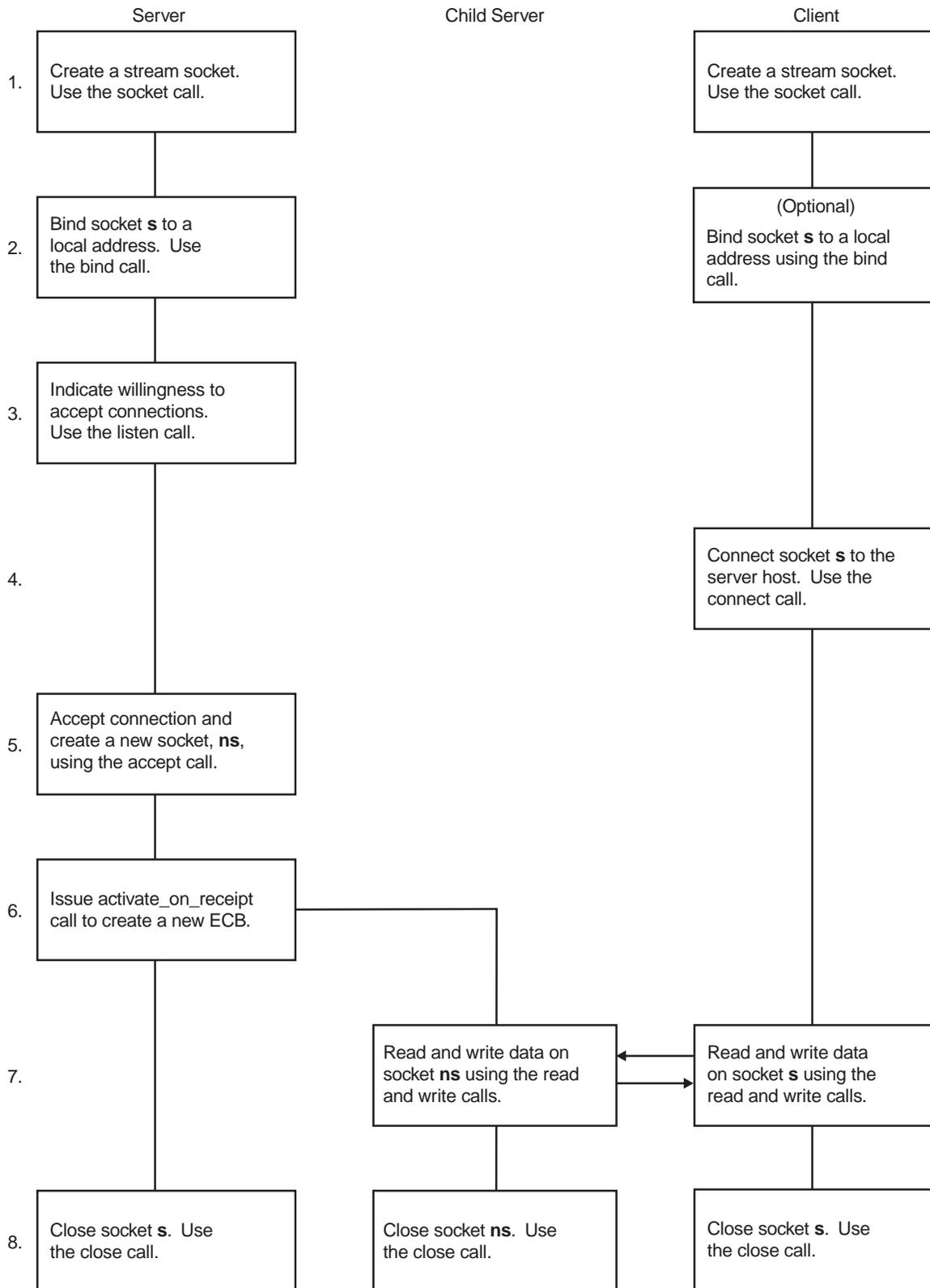


Figure 24. Using the `activate_on_receipt` Function Call

---

## Function Calls Used in a Sample UDP Session

User Datagram Protocol (UDP) is not clearly distinguished by server and client roles. The distinction is between connected and unconnected sockets. UDP uses an unconnected socket to communicate with any host. Data is sent in independent packets. Once the data has been accepted by the UDP interface, the arrival and integrity of the data is not guaranteed. Unlike connected Transmission Control Protocol (TCP) sockets, *connectionless* sockets can communicate with any server.

Figure 25 shows a general sequence of function calls used in a connectionless socket session and indicates the calls issued by the server and by the client.

The following steps correspond to the numbers in Figure 25:

1. The server and client both create a socket **s**.
2. The server uses the `bind` call to associate a local address to the socket. The client can issue an *optional* `bind` call to a local address.
3. The `sendto` and `recvfrom` calls between the client and server continue until all the data has been transferred.
4. Both the server and client end the session using the `close` call.

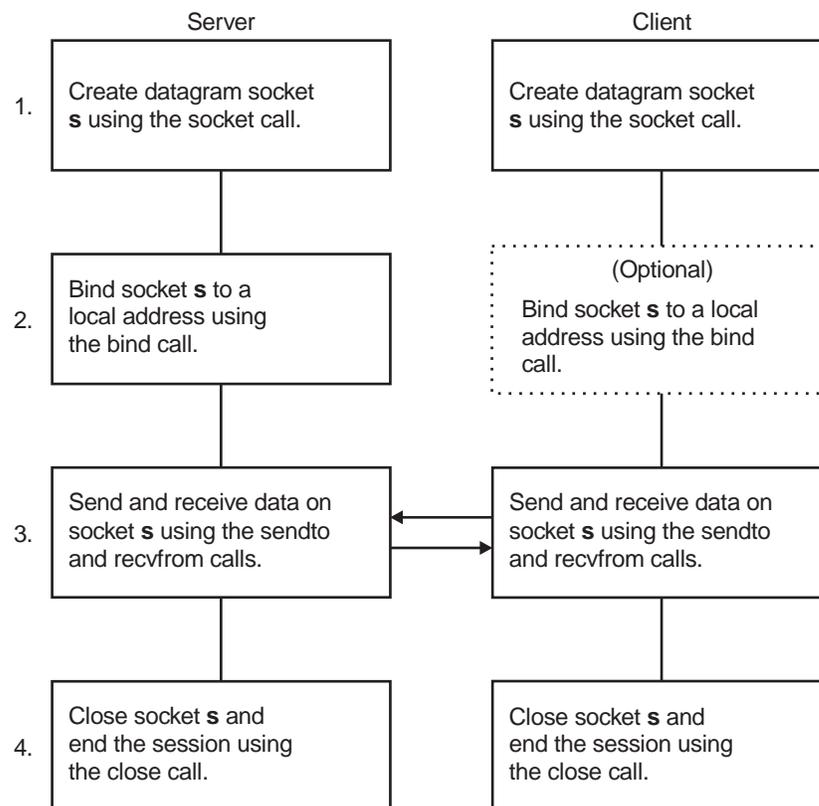


Figure 25. Sample Socket Session Using UDP Protocol

---

## Main Socket Function Calls

Following are examples of code segments showing the main socket function calls and the input and output for the calls.

1. Allocate a socket descriptor:

### socket function call

```
int socket(int domain, int type, int protocol);
:
:
int server_sock;
:
:
server_sock = socket(AF_INET, SOCK_STREAM, 0);
```

The previous example allocates socket descriptor *server\_sock* in the internet addressing family, *AF\_INET*, using a socket stream type and the default protocol TCP indicated by 0.

2. You can bind an address to a socket in two ways:
  - a. Explicitly bind a unique address to the socket:

### bind function call

```
int bind(int s, struct sockaddr *name, int namelen);
:
:
int rc;
int server_sock;
struct sockaddr_in myname;
:
:
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_port = 5001;
myname.sin_addr.s_addr = inet_addr("129.5.24.1");
:
:
rc = bind(server_sock, (struct sockaddr *) &myname,
          sizeof(myname));
```

The previous example binds socket *server\_sock* to internet address 129.5.24.1 and port 5001.

In this example, the network address and the port address were in the network byte order:

`inet_addr` was used to convert a character internet address to network byte order.

See each of these functions in the alphabetic reference section of “Socket Application Programming Interface Functions Reference” on page 137.

- b. Bind an address to the socket using a wild card:

#### bind function call

```
int bind(int s, struct * *name, int namelen);
:
:
int rc;
int server_sock;
struct *_in myname;
:
:
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_port = 5001;
myname.sin_addr.s_addr = INADDR_ANY; /* all interfaces */
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

**Note:** If a socket application with the *wildcard* option issues a bind to an offload device in a multiple offload device configuration, and the offload device is deactivated, the socket application remains active if there is another active offload device in the configuration.

3. The server indicates its readiness to accept connections from clients:

#### listen function call

```
int listen(int s, int backlog);
:
:
int rc;
int server_sock;
:
:
rc = listen(server_sock, 5);
```

This example indicates that the server is ready to accept calls, and that a maximum of 5 connect requests can be queued for the server. Additional requests are ignored.

**Note:** If the backlog is less than 0, it is set to 0. The backlog value is set in SOMAXCONN.

4. The client starts a connection request:

#### connect function call

```
int connect(int s, struct sockaddr *name, int namelen);
:
:
int rc;
int client_sock;
struct sockaddr_in servername;
:
:
memset(&servername, 0, sizeof(servername));
servername.sin_family = AF_INET;
servername.sin_port = 5001;
servername.sin_addr.s_addr = inet_addr("129.5.24.1");
:
:
rc = connect(server_sock, (struct sockaddr *) &servername,
             sizeof(servername));
```

This example connects socket *client\_sock* to the server with an address *servername*. This is the same server that was shown in the previous bind. The

client could optionally be blocked until the connection is accepted by the server. On a successful return, socket *server\_sock* is associated with the connection to the server.

5. The server accepts the client's connection request:

#### accept function call

```
int accept(int s, struct sockaddr *addr, int *addrlen);
:
:
int addrlen;
int newclient_sock;
int server_sock;
struct sockaddr client_addr;
:
:
addrlen = sizeof(client_addr);
newclient_sock = accept(server_sock, &client_addr, &addrlen);
```

When the server accepts a connection request on socket *server\_sock*, the name of the client and the length of the client name are returned, along with a new socket descriptor. The new socket descriptor is associated with the client that began the connection, and *server\_sock* is available to accept new connections.

6. The client and server transmit data in the connected state:

#### send and recv function calls

```
int send(int s, char *msg, int len, int flags);
int recv(int s, char *msg, int len, int flags);
:
:
int client_sock;
int bytes_sent;
int bytes_rcv;
char data_sent[256];
char data_rcv[256];
:
:
bytes_sent = send(client_sock, data_sent, sizeof(data_sent), 0);
:
:
bytes_rcv = recv(client_sock, data_rcv, sizeof(data_rcv), 0);
```

The previous example shows an application sending data on a connected socket and receiving data in response. The flag fields can be used to specify additional options for send or recv.

Clients and servers can use many function calls to transfer data, such as:

- The read, write, writev, send, and recv calls. However, these function calls can be used only on sockets in the connected state.
- The sendto and recvfrom function calls can be used in the unconnected state.

7. The client and server transmit data when they are in the connectionless state:

### sendto and recvfrom function calls

```
int sendto(int socket, char *buf, int buflen, int flags;
           struct sockaddr *addr, int addrlen);
int recvfrom(int socket, char *buf, int buflen, int flags;
            struct sockaddr *addr, int addrlen);
:
:
int addrlen;
int client_sock;
int bytes_sent;
int bytes_rcv;
char data_send[256];
char data_rcv[256];
struct sockaddr_in from_addr;
struct sockaddr_in to_addr;
:
:
addrlen = sizeof(struct sockaddr_in);
memset(&to_addr, 0, addrlen);
to_addr.sin_family = AF_INET;
to_addr.sin_port = 5001;
to_addr.sin_addr.s_addr = inet_addr("129.5.24.1");
:
:
bytes_sent = sendto(client_sock, data_send, sizeof(data_send), 0,
                   (struct sockaddr *)&to_addr, addrlen);
:
:
bytes_rcv = recvfrom(client_sock, data_rcv, sizeof(data_rcv),
                    0, (struct sockaddr *)&from_addr, &addrlen);
```

If the socket is **not** connected, additional socket address information must be passed to sendto and can be optionally returned from recvfrom. The caller must specify the recipient of the data or to be notified of the sender of the data.

Usually, sendto and recvfrom are used for datagram sockets; read, send, and recv are used for stream sockets.

8. Client and server can receive data using a special TPF function call called activate\_on\_receipt:

### activate\_on\_receipt function call issued from ECB 1

```
int accept(int s, struct sockaddr *addr, int *addrlen);
int activate_on_receipt(unsigned int s,
                       unsigned char *parm,
                       unsigned char *pgm);
:
:
int addrlen;
int newclient_sock;
int server_sock;
char aorparm[8];
char aorpgm[4] = "abcd";
:
:
newclient = accept(server_sock, (struct sockaddr *)0, (int *)0);
:
:
/* No parameters will be passed to the new ECB */
memset(aorparm,0,sizeof(aorparm));
rc = activate_on_receipt(newclient, aorparm, aorpgm);
```

The activate\_on\_receipt function call allows the issuing ECB to exit and activates a different ECB at program abcd. After the information has been

received, the activated program, called the child server program, must issue a read, recv, or recvfrom function call to receive the information. See “activate\_on\_receipt — Activate a Program after Data Received” on page 144 for more information.

#### read function call issued from ECB 2

```
:
abcd()
{
:
:
int bytes_rcv;
int newclient_sock;
int msg_length;
char *read_addr;
:
:
/* socket descriptor, buffer address, and message */
/* length are returned in ECB */
newclient_sock = (int)ecbptr()->ebw012;
memcpy(&read_addr,&(ecbptr()->ebw012),sizeof(read_addr));
memcpy(&msg_length,&(ecbptr()->ebw016),sizeof(msg_length));
bytes_rcv = read(newclient_sock,read_addr,msg_length);
:
:
}
```

9. Deallocate the socket descriptor:

#### close function call

```
int close(int s);
:
:
int rc;
int server_sock;
:
:
rc = close(server_sock);
```

In the previous example socket *server\_sock* is closed. The close call shuts down the socket descriptor *server\_sock* and frees up its resources.

---

# Socket Application Programming Interface Functions Reference

This chapter documents socket functions. TCP/IP support provides a set of ISO-C functions, called socket APIs, that application programs use to access the Internet. The set of interfaces that the TPF system implements is based on the industry standard.

---

## General Function Information

The functions in this section are listed alphabetically and contain the following information:

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

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

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

**Error Return** What is returned when the requested service cannot be performed. A system error with exit occurs when incorrect function parameters are specified.

**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 run. Also, if the use of a particular function affects the use of another function, that is described.

**Examples** A code segment that shows a sample function call.

**Related Information**

Where to find additional information that pertains to this function.

---

## accept — Accept a Connection Request

The `accept` function is used by a server to accept a connection request from a client.

### Format

```
#include <socket.h>
int      accept(int s,
              struct sockaddr *addr,
              int *addrlen);
```

**s** The socket descriptor. The **s** parameter is a stream socket descriptor created with the `socket` function. It is bound to an address with the `bind` function. The `listen` function marks the socket as one that accepts connections and allocates a queue to hold ending connection requests. The `listen` function allows the caller to place an upper boundary on the size of the queue.

#### addr

The socket address of the connecting client that is filled in by `accept` before it returns. The format of **addr** is determined by the domain in which the client resides. This parameter can be `NULL` if the caller is not interested in the address of the client. If the **addr** parameter is not `NULL`, it points to a **sockaddr** structure.

#### addrlen

Must initially point to an integer that contains the size, in bytes, of the storage pointed to by **addr**. On return, that integer contains the size of the data returned in the storage pointed to by **addr**. If **addr** is `NULL`, **addrlen** is ignored and can be `NULL`.

### Normal Return

A nonnegative socket descriptor indicates that the call was successful.

### Error Return

A socket descriptor of `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCACCES</b>	The socket accept user exit has rejected an incoming connection request.
<b>SOCINVAL</b>	The <code>listen</code> function was not called for socket <b>s</b> or an incorrect length was passed on the <b>addrlen</b> parameter.
<b>SOCNOBUFS</b>	There is not enough buffer space available to create the new socket. This error code is returned only for TCP/IP offload support.
<b>SOC SOCKTNOSUPPORT</b>	The <b>s</b> parameter is not of type <b>SOCK_STREAM</b> .
<b>SOCFAULT</b>	Using <b>addr</b> and <b>addrlen</b> would result in an attempt to copy the address into a protected address space.

<b>SOCWOULDBLOCK</b>	The socket <b>s</b> is in nonblocking mode and no connections are in the queue.
<b>SOCCONNABORTED</b>	The software caused a connection abend. This error code is returned only for TCP/IP offload support.
<b>EIBMIUCVERR</b>	The accept function was not successful because an error was received from the offload device. This error code is returned only for TCP/IP offload support.
<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.
<b>SOCTIMEDOUT</b>	The operation timed out. The socket is still available. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- The accept function creates a new socket descriptor with the same properties as **s** and returns it to the caller. Do not use the new socket to accept new connections. The original socket, **s**, remains available to accept more connection requests.
- The accept function accepts the first connection on its queue of pending connections. If the queue has no pending connection requests, accept blocks the caller unless **s** is in nonblocking mode. If no connection requests are queued and **s** is in nonblocking mode, accept returns -1 and sets sock\_errno to SOCWOULDBLOCK.
- The accept function is used only with **SOCK\_STREAM** sockets. There is no way to screen requesters without calling accept. The application cannot tell the system from which requesters it accepts connections. However, the caller can choose to close a connection immediately after determining the identity of the requester. If a connection request is rejected by the socket accept user exit, the accept function returns -1 and sets sock\_errno to SOCACCES.
- Check a socket for incoming connection requests by using the select for read function.
- For sockets using TCP/IP native stack support, the receive timeout value (the SO\_RCVTIMEO setsockopt option) determines how long to wait for a remote client to connect before the accept function times out.
- The accept function cannot be issued if an activate\_on\_accept call is pending for the socket. These operations are mutually exclusive.
- For sockets using TCP/IP native stack support, the socket accept connection user exit is UACC.

## accept

### Examples

Following are two examples of the `accept` function. In the first example, the caller wants to have the address of the requester returned. In the second example, the caller does not want to have the address of the requester returned.

```
#include <socket.h>
:
int newclient_sock;
int server_sock;
struct sockaddr client_addr;
int addrlen;
/* socket, bind, and listen have been called */
```

1. I want the address now:

```
addrlen = sizeof(client_addr);
newclient_sock = accept(server_sock, &client_addr, &addrlen);
```
2. I can get the address later using `getpeername`:

```
addrlen = 0;
newclient_sock = accept(server_sock, (struct sockaddr *) 0, (int *) 0);
```

### Related Information

- “`activate_on_accept` — Activate a Program When the Client Connects” on page 141
- “`bind` — Bind a Local Name to the Socket” on page 152
- “`connect` — Request a Connection to a Remote Host” on page 157
- “`getpeername` — Return the Name of the Peer” on page 168
- “`getsockname` — Return the Name of the Local Socket” on page 174
- “`listen` — Complete Binding, Create Connection Request Queue” on page 189
- “`sock_errno` — Return the Error Code Set by a Socket Call” on page 222
- “`socket` — Create an Endpoint for Communication” on page 223.

## activate\_on\_accept — Activate a Program When the Client Connects

The `activate_on_accept` function allows the issuing entry control block (ECB) to exit and activate a different ECB in the program specified when a remote client is connected.

### Format

```
#include <socket.h>
int activate_on_accept(unsigned int s,
                      unsigned char *parm,
                      unsigned char *pgm,
                      unsigned int istream);
```

**s** The socket descriptor.

#### parm

A pointer to an 8-byte field. The data in this field is saved and passed to a new ECB starting at equate EBW016. This new ECB is created when a remote client is connected.

#### pgm

A pointer to a 4-byte field that contains the name of the TPF real-time program to be activated when a remote client is connected.

#### istream

The I-stream number on which to activate the program specified by the **pgm** parameter. If the value is 0, the TPF scheduler selects the I-stream.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

Return code -1 indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, "Socket Error Return Codes" on page 365 for more information about socket errors.

Value	Description
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCINVAL</b>	The <code>listen</code> function was not called for socket <b>s</b> , or the <code>activate_on_accept</code> or <code>accept</code> function is already pending for socket <b>s</b> , or the I-stream value supplied for <code>activate_on_accept</code> was not valid.
<b>SOCINPROGRESS</b>	Socket <b>s</b> is marked nonblocking, and the connection cannot be completed immediately. The <b>SOCINPROGRESS</b> value does not indicate an error condition.
<b>SOC SOCKTNOSUPPORT</b>	Socket <b>s</b> is not a stream socket.
<b>E1052STATE</b>	The socket was closed because the system was in or cycling down to 1052 state.
<b>ESYSTEMERROR</b>	The system closed the socket because <b>pgm</b> is not a valid program name.

### Programming Considerations

- This API is available only to sockets that use TCP/IP native stack support.

## activate\_on\_accept

- This unique TPF API function can be issued instead of the accept function. When a remote client is connected, the TPF system creates a new ECB and activates the program specified by the **pgm** parameter.
- Starting at equate EBW000, data is passed to the program specified by the **pgm** parameter in the following format:

ECB Equates	Length	Description
EBW000	4	Contains the name of the program specified by the <b>pgm</b> parameter.
EBW004	4	The socket descriptor of the listener socket, which is the value of the <b>s</b> parameter.
EBW008	4	The return code, which is -1 if an error occurred; otherwise, the return code is the socket descriptor of the socket that was just accepted.
EBW012	4	The address of a <b>SOCKADDR</b> structure that contains the address of the remote client whose connection is being accepted.
EBW016	8	The data passed by the original ECB on the <b>parm</b> parameter.

- If the return code passed to **pgm** is -1, you can get the specific error code by calling `sock_errno`.

Value	Description
<b>SOCNOTSOCK</b>	Socket <b>s</b> has been cleaned up.
<b>SOCACCES</b>	The socket accept user exit rejected the connection request.
<b>SOCTIMEDOUT</b>	The operation timed out.

- The program specified by the **pgm** parameter is activated in the same subsystem as the program that issued the `activate_on_accept` call.
- If the value of the **istream** parameter is not 0, the program specified by the **pgm** parameter is activated on the I-stream specified by the **istream** parameter. If the value of the **istream** parameter is 0, the TPF scheduler selects the I-stream on which to activate the program specified by the **pgm** parameter.
- The ECB in which **pgm** is activated is assigned the current system activation number, not the activation number of the ECB that issued `activate_on_accept`.
- The receive timeout value (the `SO_RCVTIMEO` `setsockopt` option) determines how long the TPF system waits for a remote client to be connected before the `activate_on_accept` function times out.
- The `activate_on_accept` function cannot be issued if an accept call is pending for the socket. These operations are mutually exclusive.
- The `activate_on_accept` function cannot be issued if another `activate_on_accept` call is already pending for this socket.

## Examples

After accepting a connection from a new client, an `activate_on_accept` function is issued so that `server_sock` can accept the next client request.

```
#include <types.h>
#include <socket.h>
:
:
int newclient_sock;
```

```
int server_sock;
int i_stream;
u_char aoaparm[8];
u_char aoapgm[4] = "abcd";
:
:
/* No parameters will be passed to the new ECB */
memset(aoaparm,0,sizeof(aoaparm));
rc = activate_on_accept(newclient_sock,aoaparm,aoapgm) i_stream);
:
:
```

## Related Information

- “accept — Accept a Connection Request” on page 138
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223.

---

## activate\_on\_receipt — Activate a Program after Data Received

The `activate_on_receipt` function allows the issuing ECB to exit and activate a different ECB at the program specified after information has been received.

### Format

```
#include <socket.h>
int activate_on_receipt(unsigned int s,
                      unsigned char *parm,
                      unsigned char *pgm);
```

**s** The socket descriptor.

#### parm

A pointer to an 8-byte field. The data in this field is saved and passed to a new ECB starting at EBW004. This new ECB is created after information has been received.

#### pgm

A pointer to a 4-byte field that contains the name of the TPF real-time program to be activated after information has been received.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

Return code -1 indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, "Socket Error Return Codes" on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<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>	The system closed the socket because <b>pgm</b> is a program name that does not exist.
<b>SOCNOBUFS</b>	There is not enough buffer space to issue the function call. This error code is returned only for TCP/IP offload support.
<b>SOCNOTCONN</b>	The <b>s</b> socket is a stream socket, but the socket is not connected.

<b>SOCINVAL</b>	The listen function was not called for socket <b>s</b> or the read, recv, or recvfrom function is already pending for socket <b>s</b> .
<b>EIBMIUCVERR</b>	The activate_on_receipt function was not successful because an error occurred when the message was sent to the offload device. This error code is returned only for TCP/IP offload support.
<b>SOCTIMEDOUT</b>	The operation timed out. The socket is still available. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- This unique TPF API function can be issued instead of the read, recv, or recvfrom function. When the information arrives, TPF socket support creates a new ECB and activates the program specified by the **pgm** parameter.
- If TCP/IP activate on receipt load balancing (set by the AOR\_BALANCE option of the ioctl function) is set on, the ECB is created on the least busy l-stream. If load balancing is set off, the ECB is created on the l-stream of the ECB that issued the activate\_on\_receipt function.
- When the program specified by the **pgm** parameter is activated, the socket descriptor is passed in the 3 bytes at the EBROUT label in the ECB, and the address and length of the data that has been received is passed in the ECB work area fields EBW012–EBW019. The program must then issue a read or recv socket API function for stream sockets or a recvfrom socket API function for datagram sockets to receive the data.
- When receiving the data, the program can either use the address passed to it in the ECB work area or provide its own storage area to obtain the data. The length specified in the read, recvfrom, or recv function must equal the value passed to it in the ECB work area. If the value specified in the read, recv, or recvfrom function is less than the value in the ECB work area, the message is truncated. If the application program activated supplies its own buffer to read the data, the system buffer address in EBW012–EBW015 is released on the ensuing read, recv, or recvfrom function. The **MSG\_OOB** and **MSG\_PEEK** flags cannot be used for the subsequent recv or recvfrom function.
- Starting at EBW000, data is passed to the program specified by the **pgm** parameter in the following format:

ECB Equates	Length	Description
EBW000	4	Contains the name of the program specified by the <b>pgm</b> parameter. For system use only.
EBW004	8	The data passed by the original ECB pointed to by <b>parm</b> .
EBW012	4	The address of the data that was received.
EBW016	4	Length of the data that has been received by the system and should be received by the application.
EBW020	16	If a datagram socket was used, this field will contain the source address of the message.

## activate\_on\_receipt

**Note:** Information returned from an `activate_on_receipt` function is only returned to the program specified in the `activate_on_receipt` function; it is not returned to the program that issued the `activate_on_receipt`.

- If EBW012–EBW019 is equal to zero, the `activate_on_receipt` function was issued on a socket that is shut down or closed. If EBW012–EBW015 is equal to 0 and EBW016–EBW019 is equal to –1, the `activate_on_receipt` function resulted in an error return from the TCP/IP offload device. For both conditions, the application program that was activated must still issue a `read`, `recvfrom`, or `recv` socket API function to enable socket API support to complete the processing of the `activate_on_receipt` function.
- If the return code is –1, the ECB is still connected to the program. However, because it indicates a serious error, the ECB cannot issue any more socket API functions for this socket but can issue socket API functions for other sockets.
- For sockets using TCP/IP native stack support, the receive timeout value (the `SO_RCVTIMEO` setsockopt option) determines how long to wait for data to be received before the `activate_on_receipt` function times out.
- For TCP sockets using TCP/IP native stack support, the receive low-water mark (the `SO_RCVLOWAT` setsockopt option) determines the minimum amount of data that must be received before the `activate_on_receipt` function is completed. If the `activate_on_receipt` function times out, any data that was received is returned to the application even if the amount of data received is less than the receive low-water mark value.
- The `activate_on_receipt` function cannot be issued if an `activate_on_receipt_with_length`, `read`, `recv`, or `recvfrom` call is pending for the socket. These operations are mutually exclusive.
- For both TCP/IP offload support and TCP/IP native stack support, the `activate_on_receipt` function cannot be issued if another `activate_on_receipt` call is pending for this socket.

## Examples

After accepting a connection from a new client, an `activate_on_receipt` function is issued so that `server_sock` can accept the next client request.

```
#include <types.h>
#include <socket.h>
:
:
int newclient_sock;
int server_sock;
u_char aorparm[8];
u_char aorpgm[4] = "abcd";
:
:
for(;;)
{
    newclient_sock = accept(server_sock,(struct sockaddr *)0,(int)0);

    :

    /* No parameters will be passed to the new ECB */
    memset(aorparm,0,sizeof(aorparm));
    rc = activate_on_receipt(newclient_sock,aorparm,aorpgm);

    :
    :
}
```

## Related Information

- “`read` — Read Data on a Socket” on page 193
- “`recv` — Receive Data on a Connected Socket” on page 196

## **activate\_on\_receipt**

- “recvfrom — Receive Data on Connected/Unconnected Socket” on page 199
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223.

---

## activate\_on\_receipt\_with\_length — Activate a Program after Data of Specified Length Received

The `activate_on_receipt_with_length` function allows the issuing entry control block (ECB) to exit and activate a different ECB at the program specified after information with a specified length has been received.

### Format

```
#include <socket.h>
int activate_on_receipt_with_length(unsigned int s,
                                   unsigned char *parm,
                                   unsigned char *pgm,
                                   unsigned int len);
```

**s** The socket descriptor.

#### parm

A pointer to an 8-byte field. The data in this field is saved and passed to a new ECB starting at EBW004. This new ECB is created after information has been received.

#### pgm

A pointer to a 4-byte field that contains the name of the TPF real-time program to be activated after information has been received.

#### len

The length of data to be read before giving control to the new ECB.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, "Socket Error Return Codes" on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCKETNOTSOCK</b>	The <code>s</code> parameter is not a valid socket descriptor.
<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.

## activate\_on\_receipt\_with\_length

<b>SOCNOBUFS</b>	There is not enough buffer space to issue the function call. This error code is returned only for TCP/IP offload support.
<b>SOCNOTCONN</b>	The <b>s</b> socket is a stream socket, but the socket is not connected.
<b>EIBMIUCVERR</b>	The activate_on_receipt_with_length function was not successful because an error occurred when the message was sent to the offload device. This error code is returned only for TCP/IP offload support.
<b>SOCTIMEDOUT</b>	The operation timed out. The socket is still available. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- This unique TPF application programming interface (API) function can be issued instead of the read, recv, or recvfrom function. When the information arrives, TPF socket support creates a new ECB and activates the program specified by the **pgm** parameter.
- If TCP/IP activate on receipt load balancing (set by the AOR\_BALANCE option of the ioctl function) is set on, the ECB is created on the least busy l-stream. If load balancing is set off, the ECB is created on the l-stream of the ECB that issued the activate\_on\_receipt\_with\_length function.
- When the program specified by the **pgm** parameter is activated, the socket descriptor is passed in the 3 bytes at the EBROUT label in the ECB, and the address and length of the data that has been received is passed in the ECB work area fields EBW012–EBW019. The program must then issue a read or recv socket API function for stream sockets or a recvfrom socket API function for datagram sockets to receive the data.
- The number of bytes received (in EBW016) will be as many as the number of bytes of data specified by the **len** parameter. If fewer than the number of bytes requested is available, the function returns the number currently available.
- When receiving the data, the program can either use the address passed to it in the ECB work area or provide its own storage area to obtain the data. The length specified in the read, recvfrom, or recv function must equal the value passed to it in the ECB work area. If the value specified in the read, recv, or recvfrom function is less than the value in the ECB work area, the message is truncated. If the application program that is activated supplies its own buffer to read the data, the system buffer address in EBW012–EBW015 is released on the ensuing read, recv, or recvfrom function. The **MSG\_OOB** and **MSG\_PEEK** flags cannot be used for the subsequent recv or recvfrom function.
- Starting at EBW000, data is passed to the program specified by the **pgm** parameter in the following format:

ECB Equates	Length	Description
EBW000	4	Contains the name of the program specified by the <b>pgm</b> parameter. For system use only.
EBW004	8	The data passed by the original ECB pointed to by the <b>parm</b> parameter.
EBW012	4	The address of the data that was received.

## activate\_on\_receipt\_with\_length

ECB Equates	Length	Description
EBW016	4	Length of the data that has been received by the system and should be received by the application.
EBW020	16	If a datagram socket was used, this field will contain the source address of the message.

**Note:** Information returned from an `activate_on_receipt_with_length` function is only returned to the program specified in the `activate_on_receipt_with_length` function; it is not returned to the program that issued the `activate_on_receipt_with_length` function.

- If EBW012–EBW019 is equal to zero, the `activate_on_receipt_with_length` function was issued on a socket that is shut down or closed. If EBW012–EBW015 is equal to 0 and EBW016–EBW019 is equal to –1, the `activate_on_receipt_with_length` function resulted in an error return from the TCP/IP offload device. For both conditions, the application program that was activated must still issue a `read`, `recvfrom`, or `recv` socket API function to enable socket API support to complete the processing of the `activate_on_receipt_with_length` function.
- If the return code is –1, the ECB is still connected to the program. However, because it indicates a serious error, the ECB cannot issue any more socket API functions for this socket, but can issue socket API functions for other sockets.
- For sockets using TCP/IP native stack support, the receive timeout value (the `SO_RCVTIMEO` setsockopt option) determines how long to wait for data to be received before the `activate_on_receipt_with_length` function times out.
- For TCP sockets using TCP/IP native stack support, the receive low-water mark (the `SO_RCVLOWAT` setsockopt option) determines the minimum amount of data that must be received before the `activate_on_receipt_with_length` function is completed. If the `activate_on_receipt_with_length` function times out, any data that was received is returned to the application even if the amount of data received is less than the receive low-water mark value.
- The `activate_on_receipt_with_length` function cannot be issued if an `activate_on_receipt`, `read`, `recv`, or `recvfrom` call is pending for the socket. These operations are mutually exclusive.
- The `activate_on_receipt_with_length` function cannot be issued if another `activate_on_receipt_with_length` call is already pending for this socket.

## Examples

After accepting a connection from a new client, an `activate_on_receipt_with_length` function is issued so that `server_sock` can accept the next client request.

```
#include <types.h>
#include <socket.h>
...
int newclient_sock;
int server_sock;
int aor_len = 4;
u_char aorparm[8];
u_char aorpgm[4] = "abcd";
...
for(;;)
{
    newclient_sock = accept(server_sock,(struct sockaddr *)0,(int)0);
```

## activate\_on\_receipt\_with\_length

```
⋮
/* No parameters will be passed to the new ECB */
memset(aorparm,0,sizeof(aorparm));
/* read 1st 4 bytes of message. This can be useful, for */
/* example, if the application puts the message length in the */
/* first 4 bytes of message. The aor will read the 1st 4 */
/* bytes to get the message length, then issue subsequent */
/* reads for the rest of the data. */
rc = activate_on_receipt_with_length(newclient_sock,aorparm,aorpgm,aor_len);
⋮
}
```

## Related Information

- “activate\_on\_receipt — Activate a Program after Data Received” on page 144
- “read — Read Data on a Socket” on page 193
- “recv — Receive Data on a Connected Socket” on page 196
- “recvfrom — Receive Data on Connected/Unconnected Socket” on page 199
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223.

## bind — Bind a Local Name to the Socket

The `bind` function binds a unique local name to the socket with descriptor `s`.

### Format

```
#include <socket.h>
int      bind(int s,
             struct sockaddr *name,
             int namelen);
```

**s** The socket descriptor.

**name**

Pointer to a `sockaddr` structure (buffer) containing the name that is to be bound to `s`.

**namelen**

Size of the buffer pointed to by **name**, in bytes.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

The return code `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCADDRINUSE</b>	The address is already in use. See “setsockopt — Set Options Associated with a Socket” on page 216 for more information on the <code>SO_REUSEADDR</code> option.
<b>SOCADDRNOTAVAIL</b>	The address specified is not valid on this host.
<b>SOCAFNOSUPPORT</b>	The address family is not supported.
<b>SOCFAULT</b>	Using <b>name</b> and <b>namelen</b> would result in an attempt to copy the address into a protected address space. This error code is returned only for TCP/IP offload support.
<b>SOCINVAL</b>	The socket is already bound to an address. For example, you cannot bind a name to a socket that is in the <code>connected</code> state. This value is also returned if <b>namelen</b> is not the expected length.
<b>SOCNOBUFS</b>	There is not enough buffer space. This error code is returned only for TCP/IP offload support.
<b>SOCNOTSOCK</b>	The <code>s</code> parameter is not a valid socket descriptor.
<b>SOCIPNOTFOUND</b>	The TPF system could not locate the Internet Protocol (IP) table header. This error code is returned only for TCP/IP offload support.
<b>EIBMIUCVERR</b>	An error occurred when the function call was sent

	to the offload device. This error code is returned only for TCP/IP offload support.
<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>OFFLOADTIMEOUT</b>	No response was received from the offload device in a specified time period. This error code is returned only for TCP/IP offload support.

## Programming Considerations

- The bind function binds a unique local name to the socket with descriptor **s**. After calling socket, a descriptor does not have a name associated with it. The bind procedure also allows servers to specify from which network interfaces they want to receive UDP packets and TCP connection requests.
- The binding of a stream socket is not complete until a successful call to bind, listen, or connect is made. Applications using stream sockets must check the return values of bind, listen, and connect before using any function that requires a bound stream socket.
- When binding a socket using TCP/IP native stack support to all local IP addresses (that is, INADDR\_ANY is specified), the socket is bound to all IP routers that are currently active, as well as to any IP routers that are subsequently activated.

## Examples

1. Bind to a specific interface in the internet domain and make sure the sin\_zero field is cleared:

```
#include <socket.h>
:
:
int rc;
int s;
struct sockaddr_in myname;
:
:
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_port = 5001;
myname.sin_addr.s_addr = inetaddr("129.5.24.1"); /*specific interface*/
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

2. Bind to all network interfaces in the internet domain.

```
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_port = 5001;
myname.sin_addr.s_addr = INADDR_ANY; /* all interfaces */
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

## bind

3. Bind to a specific interface in the internet domain and let the system choose a port.

```
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_port = INADDR_ANY;
myname.sin_addr.s_addr = inetaddr("129.5.24.1"); /*specific interface*/
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

## Related Information

- “connect — Request a Connection to a Remote Host” on page 157
- “getsockname — Return the Name of the Local Socket” on page 174
- “htons — Translate a Short Integer” on page 181
- “listen — Complete Binding, Create Connection Request Queue” on page 189
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223.

## close — Shut Down a Socket

### Note

This description applies only to sockets. See *TPF C/C++ Language Support User's Guide* for more information about the `close` function for files.

The `close` function shuts down a socket and frees resources allocated to that socket.

### Format

```
#include <socket.h>
int      close(int s);
```

**s** The descriptor of the socket to be closed.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, "Socket Error Return Codes" on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCALREADY</b>	Socket <b>s</b> is marked nonblocking and a previous connection attempt has not been completed. This error code is returned only for TCP/IP offload support.
<b>SOCNOTCONN</b>	The socket is not connected. This error code is returned only for TCP/IP offload support.
<b>SOCNOBUFS</b>	There is not enough buffer space to satisfy request. This error code is returned only for TCP/IP offload support.
<b>EIBMIUCVERR</b>	An error occurred while the message was sent to the offload device. This error code is returned only for TCP/IP offload support.
<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.

## close

<b>ESYSTEMERROR</b>	A system error has occurred and closed the socket. This error code is returned only for TCP/IP offload support.
<b>OFFLOADTIMEOUT</b>	A response was not received from the offload device in a specified time period. This error code is returned only for TCP/IP offload support.

## Programming Considerations

The `close` function shuts down the socket associated with socket descriptor `s`, and frees resources allocated to the socket. If `s` refers to an open TCP connection, the connection is closed. If a stream socket is closed when there is input data queued, the TCP connection is reset rather than being cleanly closed.

## Examples

The following example closes `server_sock` and exits the ECB.

```
#include <socket.h>
:
:
int rc;
int server_sock;
:
:
rc = close(server_sock);
exit(0);
```

## Related Information

- “accept — Accept a Connection Request” on page 138
- “getsockopt — Return Socket Options” on page 176
- “setsockopt — Set Options Associated with a Socket” on page 216
- “socket — Create an Endpoint for Communication” on page 223.

## connect — Request a Connection to a Remote Host

The `connect` function requests a connection to a remote host.

### Format

```
#include <socket.h>
int      connect(int s,
                struct sockaddr *name,
                int namelen);
```

**s** The socket descriptor. The **s** parameter is the socket used to originate the connection request.

#### name

Pointer to a **sockaddr** structure that contains the address of the socket to which a connection will be attempted.

#### namelen

Size, in bytes, of the **sockaddr** structure pointed to by **name**.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCADDRNOTAVAIL</b>	The calling host cannot reach the specified destination.
<b>SOCAFNOSUPPORT</b>	The address family is not supported.
<b>SOCALREADY</b>	Socket <b>s</b> is marked nonblocking and a previous connection attempt has not completed.
<b>SOCCONNREFUSED</b>	The connection request was rejected by the destination host.
<b>SOCFAULT</b>	Using <b>name</b> and <b>namelen</b> would result in an attempt to copy the address into a protected address space. This error code is returned only for TCP/IP offload support.
<b>SOCINPROGRESS</b>	Socket <b>s</b> is marked nonblocking, and the connection cannot be completed immediately. The <b>SOCINPROGRESS</b> value does not indicate an error condition.
<b>SOCISCONN</b>	Socket <b>s</b> is already connected.
<b>SOCNETUNREACH</b>	You cannot get to the network from this host. This error code is returned only for TCP/IP offload support.

## connect

<b>SOCTIMEDOUT</b>	A timeout occurred before the connection was made. This error code is returned only for TCP/IP native stack support.
<b>SOCNOBUFS</b>	There is not enough buffer space to start a new connection.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCOPNOTSUP</b>	The operation is not supported on the socket. This error code is returned only for TCP/IP offload support.
<b>SOCIPNOTFOUND</b>	The TPF system could not locate the IP table header. This error code is returned only for TCP/IP offload support.
<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.
<b>EIBMIUCVERR</b>	An error occurred while the function call was being sent to the offload device. This error code is returned only for TCP/IP offload support.
<b>SOCINVAL</b>	The <b>namelen</b> parameter is not a valid length. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- For stream sockets, the `connect` call attempts to establish a connection between two sockets. For UDP sockets, the `connect` function specifies the peer for a socket.
- The `connect` function performs two tasks when called for a stream socket. First, it completes the binding necessary for a stream socket (if it has not been previously bound using the `bind` function). Second, it attempts to make a connection to another socket.
- The `connect` function on a stream socket is used by the client application to establish a connection to a server. The server must have a passive open binding. If the server is using sockets, the server must successfully call `bind` and `listen` before a connection can be accepted by the server with `accept`. Otherwise, `connect` returns `-1` and sets the error code to `SOCCONNREFUSED`.
- If **s** is in blocking mode, the `connect` function blocks the caller until the connection is set up, or until an error is received. If the socket is in nonblocking mode, `connect` returns `-1` with the error code set to `SOCINPROGRESS` if the connection can be started (no other errors occurred). For this condition, this return code does not indicate an error condition. The caller can test the completion of the connection setup by calling the `select` for write function and testing for the ability to write to the socket.

- When called for a datagram or raw socket, the connect function specifies the peer with which this socket is associated. This lets the application use data transfer calls reserved for sockets that are in the connected state. For this condition, the send and recvfrom functions are available. Stream sockets can call the connect function only once, but datagram sockets can call the connect function multiple times to change their association. Datagram sockets can end their association by connecting to an incorrect address such as the null address (all fields zeroed).
- For sockets using TCP/IP native stack support, the receive timeout value (the SO\_RCVTIMEO setsockopt option) determines how long to wait for the connection to be established before the connect function times out.

## Examples

The following example connects to a server application that has IP address 129.5.24.1 and port number 5001.

```
#include <socket.h>
:
:
int rc;
int client_sock;
struct sockaddr_in server_addr;
:
:
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = 5001;
server_addr.sin_addr.s_addr = inet_addr("129.5.24.1");
rc = connect(client_sock, (struct sockaddr *) &server_addr,
             sizeof(server_addr));
```

## Related Information

- “accept — Accept a Connection Request” on page 138
- “bind — Bind a Local Name to the Socket” on page 152
- “htons — Translate a Short Integer” on page 181
- “listen — Complete Binding, Create Connection Request Queue” on page 189
- “select — Monitor Read, Write, and Exception Status” on page 205
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223.

## gethostbyaddr — Get Host Information for IP Address

The `gethostbyaddr` function returns information about a host specified by an Internet Protocol (IP) address.

### Format

```
#include <netdb.h>
struct hostent *gethostbyaddr(char *addr,
                              int addrlen,
                              int domain);
```

#### **addr**

A pointer to an IP address in network byte order.

#### **addrlen**

The size of the Internet address in bytes.

#### **domain**

The address domain supported (AF\_INET).

### Normal Return

This function returns a pointer to a `hostent` structure for the host name specified on the call. The `netdb.h` header file defines the `hostent` structure, which contains the following elements:

Element	Description
<b>h_name</b>	Official name of the host.
<b>h_aliases</b>	Zero-terminated array of alternative names for the host.
<b>h_addrtype</b>	Type of address being returned, always set to AF_INET.
<b>h_length</b>	Length of the address in bytes.
<b>h_addr</b>	Pointer to the network address of the host in network byte order.

**Note:** Subsequent `gethostbyaddr` calls overwrite the data in the `hostent` structure.

### Error Return

A NULL pointer indicates an error. The value of `h_errno` indicates the specific error.

**Note:** 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.

Value	Description
<b>HOST_NOT_FOUND</b>	The host name specified by the <b>addr</b> parameter was not found.
<b>TRY_AGAIN</b>	The local server did not receive a response from an authorized server. Try again later.
<b>NO_RECOVERY</b>	An irrecoverable error has occurred.
<b>NO_DATA</b>	The host name is a valid name, but there is no corresponding IP address.

### Programming Considerations

The `gethostbyaddr` function tries to resolve the host Internet address through a name server if one is present.

## Examples

The following example obtains the host name associated with a given IP address.

```
#include <types.h>
#include <socket.h>
#include <netdb.h>
:
:
struct hostent *h;
struct sockaddr_in sin;
char domain[512];
sin.sin_addr.s_addr=gethostid();
h = gethostbyaddr((char *)&sin.sin_addr.s_addr,
sizeof(struct in_addr), AF_INET);
if (h!=(struct hostent *)0)
{
    strcpy(domain,h->h_name);
    printf("gethostbyaddr was successful\n");
}
else
    printf("gethostbyaddr failed\n");
```

## Related Information

“gethostname — Return Host Name” on page 166.

---

## gethostbyname — Get IP Address Information by Host Name

The `gethostbyname` function returns information about a host specified by a host name.

### Format

```
#include <netdb.h>
struct hostent *gethostbyname(char *name);
```

#### name

The name of the host being queried.

### Normal Return

This function returns a pointer to a `hostent` structure for the host name specified on the call. The `netdb.h` header file defines the `hostent` structure, which contains the following elements:

Element	Description
<b>h_name</b>	Official name of the host.
<b>h_aliases</b>	Zero-terminated array of alternative names for the host.
<b>h_addrtype</b>	Type of address being returned, always set to <code>AF_INET</code> .
<b>h_length</b>	Length of the address in bytes.
<b>h_addr</b>	Pointer to the network address of the host in network byte order.

**Note:** Subsequent `gethostbyname` calls overwrite the data in the `hostent` structure.

### Error Return

A NULL pointer indicates an error. The value of `h_errno` indicates the specific error.

**Note:** 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.

Value	Description
<b>HOST_NOT_FOUND</b>	The host name specified by the <code>name</code> parameter was not found.
<b>TRY_AGAIN</b>	The local server did not receive a response from an authorized server. Try again later.
<b>NO_RECOVERY</b>	An irrecoverable error has occurred.
<b>NO_DATA</b>	The host name is a valid name, but there is no corresponding Internet Protocol (IP) address.

### Programming Considerations

The `gethostbyname` function tries to resolve the host name through a name server if one is present.

### Examples

The following example obtains the IP address associated with a given host name.

```
#include <types.h>
#include <socket.h>
#include <netdb.h>
```

```

:
struct sockaddr whereto;
struct hostent *hp;
struct sockaddr_in *to;
char *target;
char *hostname;

memset(&whereto, 0, sizeof(struct sockaddr));
to = (struct sockaddr_in *)&whereto;
to->sin_family = AF_INET;
to->sin_addr.s_addr = inet_addr(target);
if (to->sin_addr.s_addr != -1)
    hostname = target;
else
{
    hp = gethostbyname(target);
    if (!hp)
        printf("unknown host %s\n", target);
    else
    {
        to->sin_family = hp->h_addrtype;
        memcpy(&(to->sin_addr.s_addr), hp->h_addr, hp->h_length);
        hostname = hp->h_name;
        printf("gethostbyname was successful\n");
    }
}

```

## Related Information

“gethostbyaddr — Get Host Information for IP Address” on page 160.

## gethostid — Return Identifier of Current Host

The `gethostid` function gets the unique 32-bit identifier for the current host.

### Format

```
#include <types.h>
#include <socket.h>
int      gethostid(void);
```

### Normal Return

Return code 0 indicates that the function was successful. The `gethostid` call returns the 32-bit identifier, in host byte order of the current host, which must be unique across all hosts.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCNOBUFS</b>	There is not enough buffer space to process this function. This error code is returned only for TCP/IP offload support.
<b>EIBMIUCVERR</b>	An error occurred when the function was sent to the offload device. This error code is returned only for TCP/IP offload support.
<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>OFFLOADTIMEOUT</b>	A response was not received from the offload device in a specified time period. This error code is returned only for TCP/IP offload support.
<b>SOCFAULT</b>	There are no IP addresses defined to the TPF system. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

The output of `gethostid` is either X or Y, where X is the default local IP address (see “Default Local IP Address” on page 61 for more information) and Y is the IP address of the first active and connected offload device. To determine the output of `gethostid`, consider the following:

- If TCP/IP native stack support is defined and TCP/IP offload support is not defined, the output of `gethostid` is X.
- If TCP/IP offload support is defined and TCP/IP native stack support is not defined, the output of `gethostid` is Y.
- If both TCP/IP native stack support and TCP/IP offload support are defined, and the last time that this entry control block (ECB) called USOK it picked TCP/IP native stack support, the output of `gethostid` is X.
- If both TCP/IP native stack support and TCP/IP offload support are defined, and the last time that this ECB called USOK it picked TCP/IP offload support, the output of `gethostid` is Y.

## Examples

The following example obtains the host Internet Protocol (IP) address.

```
#include <types.h>
#include <socket.h>
:
:
u_long hostid;
:
:
hostid = gethostid();
```

## Related Information

None.

## gethostname — Return Host Name

The `gethostname` function returns the host name.

### Format

```
#include <socket.h>
int      gethostname(char *name,
                    int namelen);
```

**name**

Pointer to a buffer.

**namelen**

Length of the buffer.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCNOBUFS</b>	There is not enough buffer space to process this function. This error code is returned only for TCP/IP offload support.
<b>EIBMIUCVERR</b>	An error occurred when the function was sent to the offload device. This error code is returned only for TCP/IP offload support.
<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.
<b>OFFLOADTIMEOUT</b>	A response was not received from the offload device in a specified time period. This error code is returned only for TCP/IP offload support.
<b>SOCINVAL</b>	The <b>namelen</b> parameter is not a valid length. This error code is returned only for TCP/IP native stack support.
<b>SOCFAULT</b>	Using <b>buf</b> and <b>len</b> results in an attempt to access a

protected address space. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- The output of `gethostname` is either X or Y, where X is the complex name suffixed with the CPU identifier of the TPF host and Y is the host name of the first active and connected offload device. To determine the output of `gethostname`, consider the following:
  - If TCP/IP native stack support is defined and TCP/IP offload support is not defined, the output of `gethostname` is X.
  - If TCP/IP offload support is defined and TCP/IP native stack support is not defined, the output of `gethostname` is Y.
  - If both TCP/IP native stack support and TCP/IP offload support are defined, and the last time that this entry control block (ECB) called USOK it picked TCP/IP native stack support, the output of `gethostname` is X.
  - If both TCP/IP native stack support and TCP/IP offload support are defined, and the last time that this ECB called USOK it picked TCP/IP offload support, the output of `gethostname` is Y.
- When TCP/IP native stack support is defined, the length of the buffer must be a minimum of 10 bytes.

## Examples

The following example obtains the host Internet Protocol (IP) address.

```
#include <types.h>
#include <socket.h>
:
:
int rc;
int server_sock;
u_char hostname[50];
:
rc = gethostname(&hostname, sizeof(hostname));
printf("hostname = %s\n", hostname);
```

## Related Information

“`gethostid` — Return Identifier of Current Host” on page 164.

## getpeername — Return the Name of the Peer

The `getpeername` function returns the address of the peer connected to socket `s`.

### Format

```
#include <socket.h>
int      getpeername(int s,
                    struct sockaddr *name,
                    int *namelen);
```

**s** The socket descriptor.

#### name

Pointer to the **sockaddr** buffer. On return, the buffer contains the name of the remote peer of the socket.

#### namelen

Size of the address structure pointed to by **name**, in bytes. The **namelen** parameter must be initialized to indicate the size of the space pointed to by **name** and is set to the number of bytes copied into the space before the call returns. If the buffer of the local host is too small, the peer name is truncated.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCFAULT</b>	Using <b>name</b> and <b>namelen</b> would result in an attempt to access a protected address space. This error code is returned only for TCP/IP offload support.
<b>SOCNOBUFS</b>	There is not enough buffer space. This error code is returned only for TCP/IP offload support.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCNOTCONN</b>	The socket is not in the connected state.
<b>EIBMIUCVERR</b>	An error occurred while the function was sent to the offload device. This error code is returned only for TCP/IP offload support.
<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

## getpeername

descriptor has been disconnected. The socket is still available. This error code is returned only for TCP/IP offload support.

### ESYSTEMERROR

A system error has occurred and closed the socket. This error code is returned only for TCP/IP offload support.

### SOCINVAL

The **namelen** parameter is not a valid length. This error code is returned only for TCP/IP native stack support.

### OFFLOADTIMEOUT

The offload device did not issue a response to the function call in a specified time period. This error code is returned only for TCP/IP offload support.

## Programming Considerations

- This function applies to all connected sockets for both TCP and UDP protocols.
- When TCP/IP native stack support is defined, the length passed must be a minimum of 32 bytes.

## Examples

The following example obtains the peer socket address.

```
#include <socket.h>
:
:
int addrLen;
int rc,
int newclient_sock;
int server_sock;
struct sockaddr_in client_addr;
:
:
newclient_sock = accept(server_sock, (struct sockaddr *) 0, (int) 0);
:
:
addrLen = sizeof(client_addr);
rc = getpeername(newclient_sock, (struct sockaddr *)&client_addr,
&addrLen);
```

## Related Information

- “accept — Accept a Connection Request” on page 138
- “connect — Request a Connection to a Remote Host” on page 157
- “getsockname — Return the Name of the Local Socket” on page 174
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223.

---

## getservbyname — Get Server Port by Name

The `getservbyname` function returns the port number for a specified server application name.

### Format

```
#include <netdb.h>
struct servent *getservbyname(const char *name, const char *proto);
```

**name**

The name of the server application.

**proto**

The protocol of the server application.

### Normal Return

This function returns a pointer to a `servent` structure for the server application specified on the call. The `netdb.h` header file defines the `servent` structure, which contains the following elements:

Element	Description
<code>s_name</code>	Official name of the server application.
<code>s_aliases</code>	Null pointer.
<code>s_port</code>	Port number of the server application.
<code>s_proto</code>	Protocol that the server application uses.

**Note:** Subsequent `getservbyname` or `getservbyport` calls overwrite the data in the `servent` structure.

### Error Return

A NULL pointer indicates an error.

### Programming Considerations

- The application must be defined in the TCP/IP network services database.
- This function is valid only for TCP/IP native stack support.
- The `s_name` and `s_proto` values that are returned in the `servent` structure are in uppercase.

### Examples

The following example obtains the port associated with a specified server application name.

```
#include <types.h>
#include <socket.h>
#include <netdb.h>
:
:
struct servent *appl_name;
char name[4] = "FTP";
char proto[4] = "TCP";
int port;

appl_name = getservbyname(name, proto);

if (!appl_name)
    printf("unknown application %s\n", name);
else
```



### getservbyport — Get Server Name by Port

The `getservbyport` function returns the server application name based on a specified server port number.

#### Format

```
#include <netdb.h>
struct servent *getservbyport(int port, const char *proto);
```

##### **port**

The port number of the server application.

##### **proto**

The protocol of the server application.

#### Normal Return

This function returns a pointer to a `servent` structure for the server application specified on the call. The `netdb.h` header file defines the `servent` structure, which contains the following elements:

Element	Description
<b>s_name</b>	Official name of the server application.
<b>s_aliases</b>	Null pointer.
<b>s_port</b>	Port number of the server application.
<b>s_proto</b>	Protocol that the server application uses.

**Note:** Subsequent `getservbyname` or `getservbyport` calls overwrite the data in the `servent` structure.

#### Error Return

A NULL pointer indicates an error.

#### Programming Considerations

- The application must be defined in the TCP/IP network services database.
- This function is valid only for TCP/IP native stack support.
- The `s_name` and `s_proto` values that are returned in the `servent` structure are in uppercase.

#### Examples

The following example obtains the port associated with a specified server application name.

```
#include <types.h>
#include <socket.h>
#include <netdb.h>
:
:
struct servent *appl_name;
int port;
char proto[4] = "TCP";
char *name;

port = 21;

appl_name = getservbyport(port, proto);

if (!appl_name)
```

```
        printf("unknown application %s\n", name);  
    else  
    {  
        name = appl_name->s_name;  
        printf("getservbyport was successful\n");  
    }
```

## Related Information

- “TCP/IP Network Services Database Support” on page 111
- “getservbyname — Get Server Port by Name” on page 170.

## getsockname — Return the Name of the Local Socket

The `getsockname` returns the name of the local socket.

### Normal Return

```
#include <socket.h>
int      getsockname(int s,
                    struct sockaddr *name,
                    int *namelen);
```

**s** The socket descriptor.

#### name

Address of a **sockaddr** buffer into which `getsockname` copies the local address of the socket.

#### namelen

Must initially point to an integer that contains the size, in bytes, of the storage pointed to by **name**. On return, that integer contains the size of the data returned in the storage pointed to by **name**.

### Format

Return code 0 indicates that the function was successful.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCFAULT</b>	Using <b>name</b> and <b>namelen</b> would result in an attempt to access a protected address space. This error code is returned only for TCP/IP offload support.
<b>SOCNOBUFS</b>	There is not enough buffer space. This error code is returned only for TCP/IP offload support.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<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.
<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>OFFLOADTIMEOUT</b>	The offload device did not respond to the function call in a specified time period. This error code is returned only for TCP/IP offload support.
<b>SOCNOTCONN</b>	The socket is not bound to an IP address. This error code is returned only for TCP/IP native stack support.
<b>SOCINVAL</b>	The <b>namelen</b> parameter is not a valid length. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- The `getsockname` function stores the current name for the socket specified by the **s** parameter into the structure pointed to by the **name** parameter. It returns the address to the socket that has been bound. If the socket is not bound to an address, the call returns with the family set, and the rest of the structure is set to 0.
- Stream sockets are not assigned a name until after a successful call to the `bind`, `connect`, or `accept` function.
- The `getsockname` function is often used to find the port assigned to a socket after the socket has been implicitly bound to a port. For example, an application can call `connect` without previously calling `bind`. In this case, the `connect` function completes the binding necessary by assigning a port to the socket. This assignment can be found with a call to `getsockname`.

## Examples

The following example obtains its socket address information.

```
#include <socket.h>
:
:
int addrLen;
int rc;
int server_sock;
struct sockaddr_in server_addr;
:
:
addrLen = sizeof(server_addr);
rc = getsockname(server_sock, (struct sockaddr *)&server_addr, &addrLen);
```

## Related Information

- “`accept` — Accept a Connection Request” on page 138
- “`bind` — Bind a Local Name to the Socket” on page 152
- “`connect` — Request a Connection to a Remote Host” on page 157
- “`getpeername` — Return the Name of the Peer” on page 168
- “`sock_errno` — Return the Error Code Set by a Socket Call” on page 222
- “`socket` — Create an Endpoint for Communication” on page 223.

---

## getsockopt — Return Socket Options

The `getsockopt` function gets options associated with a socket.

### Format

```
#include <socket.h>
int      getsockopt(int s,
                  int level,
                  int optname,
                  char *optval,
                  int *optlen);
```

**s** The socket descriptor.

#### level

Level for which the option is set. Use the value **SOL\_SOCKET**.

#### optname

Name of a specified socket option. Use one of the following values:

##### SO\_BROADCAST

Returns the value of the broadcast messages. Enabling this option lets the application send broadcast messages over **s** if the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.

##### SO\_DONTROUTE

Returns the value of the outgoing messages. When you enable this option, outgoing messages bypass the standard routing algorithm and are directed to the appropriate network interface according to the network portion of the destination address. When enabled, this option lets you send packets only to directly connected networks (networks for which the host has an interface). This option has no meaning for stream sockets.

##### SO\_ERROR

Returns any pending error on the socket and clears the error status. You can use it to check for asynchronous errors on connected datagram sockets or for other asynchronous errors (errors that are not returned explicitly by one of the socket calls).

##### SO\_KEEPALIVE

Sends probes on idle sockets to verify that the socket is still active. This option has meaning only for stream sockets.

##### SO\_LINGER

Waits to complete the `close` function if data is present. When you enable this option and there is unsent data present when `close` is called, the calling application is blocked during the `close` function until the data is transmitted or the connection has timed out. The `close` function returns without blocking the caller. This option has meaning only for stream sockets.

##### SO\_OOBINLINE

Toggles reception of out-of-band data. Enabling this option causes out-of-band data to be placed in the normal data input queue as it is received, making it available to `recvfrom` and `recv` without having to specify the **MSG\_OOB** flag in those calls. Disabling this option causes out-of-band data to be placed in the priority data input queue as it is received, making it available to `recvfrom` and `recv` only by specifying the **MSG\_OOB** flag in these functions. This option has meaning only for stream sockets.

**SO\_RCVBUF**

Returns the size of the receive buffer. This option has meaning only for sockets that are using TCP/IP native stack support.

**SO\_RCVLOWAT**

Returns the receive buffer low-water mark, which is the minimum amount of data that must be received before a read, recv, recvfrom, activate\_on\_receipt, activate\_on\_receipt\_with\_length, or activate\_on\_receipt\_with\_length function is completed successfully. This option has meaning only for TCP sockets that are using TCP/IP native stack support.

**SO\_RCVTIMEO**

Returns the receive timeout value, which is how long the system will wait for a read, recv, recvfrom, activate\_on\_receipt, activate\_on\_receipt\_with\_length, accept, activate\_on\_accept, or connect function to be completed successfully before timing out the operation. A returned value of 0 indicates the system will not time out. This option has meaning only for sockets that are using TCP/IP native stack support.

**SO\_REUSEADDR**

Toggles local address reuse. Enabling this option allows local addresses that are already in use to be bound. This changes the normal algorithm used in the bind function. At connect time, the system checks that no local address and port have the same remote address and port, and returns error code SOCADDRINUSE if the association already exists.

**SO\_SNDBUF**

Allows you to set the size of the send buffer to a value to suit your application needs.

**SO\_SNDLOWAT**

Returns the send buffer low-water mark, which is the minimum amount of space that must be available in the send buffer to allow a select for write function to be processed. This option has meaning only for sockets that are using TCP/IP native stack support.

**SO\_SNDTIMEO**

Returns the send timeout value, which is how long the system will wait for a send, sendto, write, or writev function to be completed before timing out the operation. A returned value of 0 indicates the system will not time out. This option has meaning only for sockets that are using TCP/IP native stack support.

**SO\_TYPE**

Returns the type of the socket. On return, the integer pointed to by **optval** is set to one of the following values:

- **SOCK\_STREAM**
- **SOCK\_DGRAM**
- **SOCK\_RAW**.

**optval**

Pointer to option data. The **optval** and **optlen** parameters return data used by the particular get command. The **optval** parameter points to a buffer that is to receive the data requested by the get command.

**optlen**

Pointer to the length of the option data. The **optlen** parameter points to the size

## getsockopt

of the buffer pointed to by the **optval** parameter. It must be initially set to the size of the buffer before calling `getsockopt`. On return, it is set to the actual size of the data returned.

## Normal Return

Return code 0 indicates that the function was successful.

## Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCFAULT</b>	Using <b>optval</b> and <b>optlen</b> parameters would result in an attempt to copy the address into a protected address space. This error code is returned only for TCP/IP offload support.
<b>SOCNOPROTOOPT</b>	The <b>optname</b> parameter is not recognized, or the level parameter is not <b>SOL_SOCKET</b> .
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCNOBUFS</b>	There is not enough buffer space available to process the message. 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.
<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>OFFLOADTIMEOUT</b>	The offload device did not respond to the function call in a specified time period. This error code is returned only for TCP/IP offload support.
<b>SOCINVAL</b>	The socket type is incorrect for the option passed. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- The `getsockopt` function gets options associated with a socket.
- When manipulating socket options, you must specify the name of the option and the level at which the option resides.
- When TCP/IP native stack support is defined, the length passed by the `optlen` parameter must be a minimum of 4 bytes.
- All of the socket-level options except **SO\_LINGER** expect `optval` to point to an integer and `optlen` to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is 0, the option is disabled. The **SO\_LINGER** option expects `optval` to point to a `linger` structure. The `linger` structure is defined in the following example:

```
struct linger
    int    l_onoff;           /* option on/off */
    int    l_linger;        /* linger time */
    :
    :
```

- The `l_onoff` is set to 0 if the **SO\_LINGER** option is being disabled. A nonzero value enables the option. The `l_linger` field specifies the amount of time to wait before completing a `close` when there is still data to be sent.
- The `getsockopt` function is rejected if the socket and the `bind` function has not been issued.

## Examples

The following example obtains out-of-band information.

```
#include <socket.h>
:
int optval;
int optlen;
int rc;
int server_sock;
:
/* Is out of band data in the normal input queue? */
optlen = sizeof(optval);
rc = getsockopt(server_sock, SOL_SOCKET, SO_OOBINLINE,
               (char *)&optval, &optlen);
if (rc == 0)
    if (optlen == sizeof(int))
    {
        if (optval)
            /* yes it is in the normal queue */
        else
            /* no it is not */
    }
```

## Related Information

- “`setsockopt` — Set Options Associated with a Socket” on page 216
- “`sock_errno` — Return the Error Code Set by a Socket Call” on page 222
- “`socket` — Create an Endpoint for Communication” on page 223.

### htonl — Translate a Long Integer

The `htonl` function translates a long integer from host byte order to network byte order.

#### Format

```
#include <types.h>
#include <socket.h>
u_long  htonl(u_long a);
```

**a** Unsigned long integer to be put into network byte order.

#### Normal Return

Returns the translated long integer.

#### Error Return

None.

#### Programming Considerations

None.

#### Examples

The following example converts the information from host byte order to network byte order.

```
#include <types.h>
#include <socket.h>
:
:
u_long a;
u_long n;
:
n = htonl(a);
```

#### Related Information

- “htons — Translate a Short Integer” on page 181
- “ntohl — Translate a Long Integer” on page 191
- “ntohs — Translate a Short Integer” on page 192.

---

## htons — Translate a Short Integer

The `htons` function translates a short integer from host byte order to network byte order.

### Format

```
#include <types.h>
#include <socket.h>
u_short  htons(u_short a);
```

**a** Unsigned short integer to be put into network byte order.

### Normal Return

Returns the translated short integer.

### Error Return

None.

### Programming Considerations

None.

### Examples

The following example converts the information from host byte order to network byte order.

```
#include <types.h>
#include <socket.h>
:
:
u_short a;
u_short n;
:
n = htons(a);
```

### Related Information

- “`htonl` — Translate a Long Integer” on page 180
- “`ntohl` — Translate a Long Integer” on page 191
- “`ntohs` — Translate a Short Integer” on page 192.

---

## inet\_addr — Construct Internet Address from Character String

The `inet_addr` function interprets character strings representing numbers expressed in standard dotted decimal notation and returns numbers suitable for use as an internet address.

### Format

```
#include <types.h>
#include <socket.h>
u_long  inet_addr(char *cp);
```

**cp** A character string in standard dotted decimal notation.

### Normal Return

The internet address is returned in network byte order.

### Error Return

A return code equal to `-1` indicates a character string that is not valid.

### Programming Considerations

- Values specified in standard dotted decimal notation take one of the following forms:
  - a.b.c.d
  - a.b.c
  - a.b
  - a
- When you specify a four-part address, each part is interpreted as a byte of data and assigned, from left to right, to one of the 4 bytes of an internet address.
- When you specify a three-part address, the last part is interpreted as a 16-bit quantity and placed in the 2 rightmost bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as `128.net.host`.
- When you specify a two-part address, the last part is interpreted as a 24-bit quantity and placed in the 3 rightmost bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as `net.host`.
- When you specify a one-part address, the value is stored directly in the network address space without any rearrangement of its bytes.
- Numbers supplied as address parts in standard dotted decimal notation can be decimal, hexadecimal, or octal. Numbers are interpreted in C language syntax. A leading `0x` implies hexadecimal; a leading `0` implies octal. A number without a leading `0` implies decimal.

### Examples

The following example converts character IP address to network byte order.

```
#include <types.h>
#include <socket.h>
:
:
struct sockaddr_in server_addr;
:
:
servername.sin_addr.s_addr = inet_addr("129.5.24.1");
```

## Related Information

None.

---

## inet\_ntoa — Return Pointer to a String in Dotted Decimal Notation

The `inet_ntoa` function returns a pointer to a string in dotted decimal notation.

### Format

```
#include <types.h>
#include <socket.h>
char *inet_ntoa(struct in_addr in);
```

**in** The host Internet address.

### Normal Return

This function returns a pointer to a string expressed in dotted decimal notation. The call accepts an Internet address expressed as a 32-bit quantity in network byte order and returns a string expressed in dotted decimal notation. Storage pointed to exists on an entry control block (ECB) basis and is overwritten by subsequent calls.

### Error Return

None.

### Programming Considerations

None.

### Examples

The following example returns a pointer to the input address in dotted decimal notation.

```
#include <types.h>
#include <socket.h>
:
:
struct in_addr in;
char *inetadd;

in = 0x975C645;
inetadd = inet_ntoa(in);
printf("IP address is %s\n",inetadd);
```

### Related Information

“`inet_addr` — Construct Internet Address from Character String” on page 182.

## ioctl — Perform Special Operations on Socket

The `ioctl` function performs special operations on socket descriptor `s`.

### Format

```
#include <types.h>
#include <socket.h>
#include <ioctl.h>
int      ioctl(int s,
               int cmd,
               char *arg);
```

**s** The socket descriptor.

#### cmd

Command to perform. Use one of the following values:

#### AOR\_BALANCE

Sets TCP/IP activate on receipt load balancing on or off for a socket. If load balancing is set on, the ECB is created on the least busy l-stream when an `activate_on_receipt` or `activate_on_receipt_with_length` function is completed. If load balancing is set off, the ECB is created on the l-stream of the ECB that issued the `activate_on_receipt` or `activate_on_receipt_with_length` function request. The **arg** parameter is a pointer to an integer. If the integer is nonzero, load balancing is set on; otherwise, load balancing is set off.

#### Notes:

1. TCP/IP activate on receipt load balancing can be set on only for applications that run on any l-stream.
2. TCP/IP activate on receipt load balancing is only available for sockets that use TCP/IP native stack support.
3. The `AOR_BALANCE` parameter is unique to the TPF 4.1 system.

#### FIONBIO

Sets or clears nonblocking input/output for a socket. The **arg** parameter is a pointer to an integer. If the integer is 0, nonblocking input/output on the socket is cleared. Otherwise, the socket is set for nonblocking input/output.

#### FIONREAD

Gets the number of immediately readable bytes for the socket. The **arg** parameter is a pointer to an integer. Sets the value of the integer to the number of immediately readable characters for the socket.

#### SIOCADDRT

Adds a routing table entry. The **arg** parameter is a pointer to a `rtrentry` structure as defined in `ioctl.h`. The routing table entry, passed as an argument, is added to the routing tables. This option is for IBM use only.

#### SIOCATMARK

Queries whether the current location in the data input is pointing to out-of-band data. The **arg** parameter is a pointer to an integer. Sets the argument to 1 if the socket points to a mark in the data stream for out-of-band data. Otherwise, sets the argument to 0.

#### SIOCDELRT

Deletes a routing table entry. The **arg** parameter is a pointer to a `rtrentry` structure. If it exists, the routine table entry, passed as an argument, is deleted from the routing tables. This option is for IBM use only.

## ioctl

### **SIOCGIFADDR**

Gets the network interface address. The **arg** parameter is a pointer to an `ifreq` structure, as defined in `ioctl.h`. The interface address is returned in the field in the `ifreq` structure that has the result.

### **SIOCGIFBRDADDR**

Gets the network interface broadcast address. The **arg** parameter is a pointer to an `ifreq` structure as defined in `ioctl.h`. The interface broadcast address is returned in the argument.

### **SIOCGIFCONF**

Gets the network interface configuration. The **arg** parameter is a pointer to an `ifconf` structure as defined in `ioctl.h`. The interface configuration is returned in the argument. The length must be at least 32 bytes.

### **SIOCGIFDSTADDR**

Gets the network interface destination address. The **arg** parameter is a pointer to an `ifreq` structure as defined in `ioctl.h`. The interface destination (point-to-point) address is returned in the argument.

### **SIOCGIFFLAGS**

Gets the network interface flags. The **arg** parameter is a pointer to an `ifreq` structure as defined in `ioctl.h`. The interface flags are returned in the argument.

### **SIOCGIFMETRIC**

Gets the network interface routing metric. The **arg** parameter is a pointer to an `ifreq` structure as defined in `ioctl.h`. The interface routine metric is returned in the argument. This option is for IBM use only.

### **SIOGIFNETMASK**

Gets the network interface network mask. The **arg** parameter is a pointer to an `ifreq` structure as defined in `ioctl.h`. The interface network mask is returned in the argument.

### **SIOCSIFDSTADDR**

Sets the network interface destination address. The **arg** parameter is a pointer to an `ifreq` structure as defined in `ioctl.h`. Sets the interface destination (point-to-point) address to the value passed in the argument. This option is for IBM use only.

### **SIOCSIFFLAGS**

Sets the network interface flags. The **arg** parameter is a pointer to an `ifreq` structure as defined in `ioctl.h`. Sets the interface flags to the values passed in the argument. This option is for IBM use only.

### **SIOCSIFMETRIC**

Sets the network interface routing metric. The **arg** parameter is a pointer to an `ifreq` structure as defined in `ioctl.h`. Sets the interface routing metric to the value passed in the argument. This option is for IBM use only.

### **TPF\_NOSWEEP**

Skips the socket sweeper processing for this socket. The socket sweeper facility monitors open sockets and closes those that are idle. An idle socket is one that has had no socket application programming interfaces (APIs) issued recently by any application. The **arg** parameter is a pointer to an integer. If the integer is any nonzero value, the socket sweeper facility will skip monitoring the socket descriptor supplied by the **s** parameter for socket API activity. If the **arg** parameter is set to zero, the socket sweeper monitors the socket descriptor.

**Note:** The TPF\_NOSWEEP parameter is unique to the TPF 4.1 system.

**arg**

Pointer to the data associated with **cmd**, and its format depends on the command that is requested.

## Normal Return

Return code 0 indicates that the function was successful.

## Error Return

A return code equal to  $-1$  indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCOPNOTSUP</b>	The operation is not supported on the socket.
<b>SOCNOTCONN</b>	The socket is not connected.
<b>SOCINVAL</b>	The request is not valid or not supported or the buffer passed was not the required minimum length.
<b>SOCFAULT</b>	Using <b>buf</b> and <b>len</b> would result in an attempt to access a protected address space. This error code is returned only for TCP/IP native stack support.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCNOBUFS</b>	There is not enough buffer space available to process the message. 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.
<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>OFFLOADTIMEOUT</b>	The offload device did not respond to the function call in a specified time period. This error code is returned only for TCP/IP offload support.

## ioctl

### Programming Considerations

- The operating characteristics of sockets can be controlled with the `ioctl` function. The operations to be controlled are determined by the **cmd** parameter. The **arg** parameter is a pointer to data associated with the particular command.
- For sockets using TCP/IP native stack support, the `TPF_NOSWEEP` parameter only affects socket sweeper processing based on application idle timeouts. Sockets that issue this **cmd** parameter value can still be swept because of network idle timeouts.
- `TPF_NOSWEEP` has no effect when the TPF socket sweeper facility is not active. `ZNKEY SOCKSWP` displays the current sweep timeout; a timeout of 0 indicates the sweeper is not active. When the socket sweeper facility is activated, the latest value for `TPF_NOSWEEP` determines whether the socket sweeper monitors the socket descriptor.
- Open sockets are subject to socket sweeper monitoring by default.

### Examples

The following example sets `server_sock` to be in nonblocking mode.

```
#include <types.h>
#include <socket.h>
#include <ioctl.h>
:
:
int dontblock;
int rc;
int server_sock
:
:
/* Place the socket into nonblocking mode */
dontblock = 1;
rc = ioctl(server_sock, FIONBIO, (char *) &dontblock);
```

### Related Information

“`sock_errno` — Return the Error Code Set by a Socket Call” on page 222.

## listen — Complete Binding, Create Connection Request Queue

The `listen` function completes the binding necessary for a socket and creates a connection request queue for incoming requests.

### Format

```
#include <socket.h>
int      listen(int s,
               int backlog);
```

**s** The socket descriptor.

#### backlog

Maximum length for the queue of pending connections. If **backlog** is less than 0, its value is set to 0. For sockets using TCP/IP offload support, if **backlog** is greater than `SOMAXCONN` or 5, its value is set to `SOMAXCONN`. For sockets using TCP/IP native stack support, the maximum value for **backlog** is 32 767. If the value passed is greater than 32 767, the value is set to 32 767.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCOPNOTSUP</b>	The <b>s</b> parameter is not a socket descriptor that supports the <code>listen</code> function.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCINVAL</b>	The socket is not in the correct state for listening.
<b>EIBMIUCVERR</b>	An error occurred when the function call was sent to the offload device. This error code is returned only for TCP/IP offload support.
<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>OFFLOADTIMEOUT</b>	The offload device did not respond to the function

## listen

call in the requested time period. This error code is returned only for TCP/IP offload support.

## Programming Considerations

- The `listen` function applies only to stream sockets. The function performs the following tasks:
  - It completes the binding necessary for socket `s` if `bind` has not been called for `s`.
  - It creates a connection request queue, which is the length of the **backlog** parameter, to queue incoming connection requests. After the queue is full, additional connection requests are ignored.
- The `listen` function indicates a readiness to accept client connection requests. This function transforms an active socket into a passive socket. Once called, `s` can never be used as an active socket to start connection requests. Calling `listen` is the third of four steps that a server performs to accept a connection. This function is called after allocating a stream socket with `socket`, and after binding a name to `s` with `bind`. The `listen` function must be called before calling `accept`.

## Examples

The following example sets up itself to be a server and it also creates a client request queue of size 5.

```
#include <socket.h>
:
:
int rc;
int server_sock;
:
rc = listen(server_sock, 5);
```

## Related Information

- “accept — Accept a Connection Request” on page 138
- “bind — Bind a Local Name to the Socket” on page 152
- “connect — Request a Connection to a Remote Host” on page 157
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223.

---

## ntohl — Translate a Long Integer

The `ntohl` function translates a long integer from network byte order to host byte order.

### Format

```
#include <types.h>
#include <socket.h>
u_long  ntohl(u_long a);
```

**a** Unsigned long integer to be put into host byte order.

### Normal Return

Returns the translated long integer.

### Error Return

None.

### Programming Considerations

None.

### Examples

The following example converts the information from network byte order to host byte order.

```
#include <types.h>
#include <socket.h>
:
:
u_long a;
u_long n;
:
n = ntohl(a);
```

### Related Information

- “htons — Translate a Short Integer” on page 181
- “htonl — Translate a Long Integer” on page 180
- “ntohs — Translate a Short Integer” on page 192.

## ntohs — Translate a Short Integer

The ntohs function translates a short integer from network byte order to host byte order.

### Format

```
#include <types.h>
#include <socket.h>
u_short ntohs(u_short a);
```

**a** Unsigned short integer to be put into host byte order.

### Normal Return

Returns the translated short integer.

### Error Return

None.

### Programming Considerations

None.

### Examples

The following example converts the information from network byte order to host byte order.

```
#include <types.h>
#include <socket.h>
:
:
u_short a;
u_short n;
:
:
ntohs(a);
```

### Related Information

- “htonl — Translate a Long Integer” on page 180
- “htons — Translate a Short Integer” on page 181
- “ntohl — Translate a Long Integer” on page 191.

## read — Read Data on a Socket

### Note

This description applies only to sockets. See *TPF C/C++ Language Support User's Guide* for more information about the read function for files.

The read function reads data on a socket with descriptor **s** and stores it in a buffer.

### Format

```
#include <socket.h>
int      read(int s,
             char *buf,
             int len);
```

**s** The socket descriptor.

### buf

Pointer to the buffer that receives the data.

### len

Length, in bytes, of the buffer pointed to by the **buf** parameter. The maximum amount of data that can be received is 32 768 bytes.

### Normal Return

If successful, the number of bytes copied into the buffer is returned. If an end-of-file condition is received or the connection is closed, 0 is returned.

### Error Return

A return code equal to  $-1$  indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, "Socket Error Return Codes" on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCFAULT</b>	Using <b>buf</b> and <b>len</b> results in an attempt to access a protected address space.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCWOULDBLOCK</b>	The <b>s</b> parameter is in nonblocking mode and no data is available to read.
<b>SOCNOTCONN</b>	The socket is not connected.
<b>SOCNOBUFS</b>	There is not enough space available to process the function call. This error code is returned only for TCP/IP offload support.
<b>EIBMIUCVERR</b>	An error occurred when the function call was sent to the offload device. This error code is returned only for TCP/IP offload support.
<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

## read

descriptor have been disconnected. The socket is closed. This error code is returned only for TCP/IP offload support.

### EINACTWS

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.

### ESYSTEMERROR

A system error has occurred and closed the socket.

### SOCTIMEDOUT

The operation timed out. The socket is still available. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- This function applies only to connected sockets.
- This function returns up to the number of bytes of data specified by the **len** parameter. If fewer than the number of bytes requested is available, the function returns the number currently available.
- If data is not available for the sockets **s**, and **s** is in blocking mode, the read function blocks the caller until data arrives. If data is not available, and **s** is in nonblocking mode, read returns a **-1** and sets `sock_errno` to `SOCWOULDBLOCK`.
- For sockets using TCP/IP native stack support, the receive timeout value (the `SO_RCVTIMEO` `setsockopt` option) determines how long to wait for data to be received before the read function times out.
- For TCP sockets using TCP/IP native stack support, the receive low-water mark (the `SO_RCVLOWAT` `setsockopt` option) determines the minimum amount of data that must be received before the read function is completed. If the read function times out, any data that was received is returned to the application even if the amount of data received is less than the receive low-water mark value.
- The read function cannot be issued if an `activate_on_receipt` or `activate_on_receipt_with_length` call is pending for the socket. These operations are mutually exclusive.

## Examples

After the server accepts a client connection, it reads in a message from its client.

```
#include <socket.h>
:
:
int rc;
int newclient_sock;
int server_sock;
char recv_client_msg[100];
:
:
newclient_sock = accept(server_sock, (struct sockaddr *) 0, (int) 0);
rc = read(newclient_sock,recv_client_msg,sizeof(recv_client_msg));
```

## Related Information

- “`activate_on_receipt` — Activate a Program after Data Received” on page 144
- “`connect` — Request a Connection to a Remote Host” on page 157
- “`getsockopt` — Return Socket Options” on page 176
- “`ioctl` — Perform Special Operations on Socket” on page 185
- “`recv` — Receive Data on a Connected Socket” on page 196

- “recvfrom — Receive Data on Connected/Unconnected Socket” on page 199
- “select — Monitor Read, Write, and Exception Status” on page 205
- “send — Send Data on a Connected Socket” on page 206
- “sendto — Send Data on an Unconnected Socket” on page 212
- “setsockopt — Set Options Associated with a Socket” on page 216
- “socket — Create an Endpoint for Communication” on page 223
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “write — Write Data on a Connected Socket” on page 226.

---

## recv — Receive Data on a Connected Socket

The `recv` function receives data on a socket with descriptor `s` and stores it in a buffer.

### Format

```
#include <socket.h>
int      recv(int s,
             char *buf,
             int len,
             int flags);
```

**s** The socket descriptor.

#### buf

Pointer to the buffer that receives the data.

#### len

Length, in bytes, of the buffer pointed to by the **buf** parameter. The maximum number of bytes that can be received is 32 768.

#### flags

Must be set to 0 or one or more of the following flags. If you specify more than one flag, use the logical OR operator (`|`) to separate them:

##### MSG\_OOB

Reads any out-of-band data on the socket.

##### MSG\_PEEK

Peeks at the data present on the socket; the data is returned but not consumed, so a later receive operation sees the same data.

### Normal Return

If successful, the function returns the length, in bytes, of the message or datagram.

In addition, the number of bytes copied into the buffer is returned. If an end-of-file condition is received or the connection is closed, 0 is returned.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCFAULT</b>	Using <b>buf</b> and <b>len</b> would result in an attempt to access a protected address space.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCWOULDBLOCK</b>	The <b>s</b> parameter is in nonblocking mode, and no data is available to read.
<b>SOCNOBUFS</b>	There is not enough space available to process the function call. This error code is returned only for TCP/IP offload support.

<b>SOCNOTCONN</b>	A stream socket was used to issue the <code>recv</code> function, and the socket was not connected.
<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.
<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.
<b>SOCTIMEDOUT</b>	The operation timed out. The socket is still available. This error code is returned only for TCP/IP native stack support.
<b>SOCINVAL</b>	The <code>MSG_OOB</code> option was specified for a socket other than a stream socket or the <code>MSG_OOB</code> option was specified, but out-of-band data is queued inline for this socket. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- This function applies only to connected sockets.
- This function returns up to **len** bytes of data. If there are fewer than the requested number of bytes available, the function returns the number currently available.
- If data is not available for socket **s**, and **s** is in blocking mode, the `recv` function blocks the caller until data arrives. If data is not available, and **s** is in nonblocking mode, `recv` returns a `-1` and sets `sock_errno` to `SOCWOULDBLOCK`.
- If a `bind` has not yet been issued to the socket, a `bind` is issued on behalf of the application for a non-stream socket.
- For sockets using TCP/IP native stack support, the receive timeout value (the `SO_RCVTIMEO` `setsockopt` option) determines how long to wait for data to be received before the `recv` function times out.
- For TCP sockets using TCP/IP native stack support, the receive low-water mark (the `SO_RCVLOWAT` `setsockopt` option) determines the minimum amount of data that must be received before the `recv` function is completed. If the `recv` function times out, any data that was received is returned to the application even if the amount of data received is less than the receive low-water mark value.
- The `recv` function cannot be issued if an `activate_on_receipt` or `activate_on_receipt_with_length` call is pending for the socket. These operations are mutually exclusive.

## Examples

The following example reads in 1 byte of out-of-band data.

## recv

```
#include <socket.h>
:
:
int rc;
int server_sock;
char oob_data;
:
:
rc = recv(server_sock,oob_data,sizeof(oob_data),MSG_OOB);
if (rc > 0)
{
/* Process the oob data from the sender */
:
:
}
```

## Related Information

- “activate\_on\_receipt — Activate a Program after Data Received” on page 144
- “getsockopt — Return Socket Options” on page 176
- “ioctl — Perform Special Operations on Socket” on page 185
- “read — Read Data on a Socket” on page 193
- “recvfrom — Receive Data on Connected/Unconnected Socket” on page 199
- “select — Monitor Read, Write, and Exception Status” on page 205
- “send — Send Data on a Connected Socket” on page 206
- “sendto — Send Data on an Unconnected Socket” on page 212
- “setsockopt — Set Options Associated with a Socket” on page 216
- “socket — Create an Endpoint for Communication” on page 223
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “write — Write Data on a Connected Socket” on page 226.

---

## recvfrom — Receive Data on Connected/Unconnected Socket

The `recvfrom` function receives data on a socket with descriptor `s` and stores it in a buffer.

### Format

```
#include <socket.h>
int      recvfrom(int s,
                 char *buf,
                 int len,
                 int flags,
                 struct sockaddr *name,
                 int *namelen);
```

**s** The socket descriptor.

#### buf

Pointer to the buffer that receives the data.

#### len

Length, in bytes, of the buffer pointed to by the **buf** parameter. The maximum number of bytes that can be received is 32 768.

#### flags

Must be set to 0 or one or more of the following flags. If you specify more than one flag, use the logical OR operator (`|`) to separate them:

##### MSG\_OOB

Reads any out-of-band data on the socket.

##### MSG\_PEEK

Peeks at the data present on the socket; the data is returned but not consumed so a later receive operation sees the same data.

#### name

This is a pointer to a socket address from which data is received.

#### namelen

Pointer to the size of **name** in bytes.

### Normal Return

If successful, the function returns the length, in bytes, of the message or datagram.

If an end-of-file condition is received or the connection is closed, 0 is returned.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCFAULT</b>	Using <b>buf</b> and <b>len</b> would result in an attempt to access a protected address space.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCWOULDBLOCK</b>	The <b>s</b> parameter is in nonblocking mode and no data is available to read.

## recvfrom

<b>SOCNOBUFS</b>	There is not enough space available to process the function call. This error code is returned only for TCP/IP offload support.
<b>SOCNOTCONN</b>	A stream socket was used to issue the <code>recvfrom</code> function, and the socket was not connected.
<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.
<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.
<b>SOCTIMEDOUT</b>	The operation timed out. The socket is still available. This error code is returned only for TCP/IP native stack support.
<b>SOCINVAL</b>	The value of the <b>namelen</b> parameter is not valid, the <code>MSG_OOB</code> option was specified for a socket that is not a stream socket, or the <code>MSG_OOB</code> option was specified, but out-of-band data is queued inline for this socket. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- The `recvfrom` function receives data on a socket with descriptor **s** and stores it in the caller's buffer.
- If the name is nonzero, the source address of the message is returned. The **namelen** parameter is first initialized by the caller to the size of the buffer associated with **name**; on return, it is modified to indicate the actual number of bytes stored there.
- The `recvfrom` function returns the length of the incoming message or data. If a message is too long to fit in the supplied buffer, the message is truncated. If a message is not available at the socket with descriptor **s**, the `recvfrom` function waits for a message to arrive and blocks the caller unless the socket is in nonblocking mode. See “`ioctl` — Perform Special Operations on Socket” on page 185 for a description of how to set nonblocking mode.
- If a `bind` has not yet been issued to the socket, a `bind` is issued on behalf of the application for a non-stream socket.
- For sockets using TCP/IP native stack support, the receive timeout value (the `SO_RCVTIMEO` `setsockopt` option) determines how long to wait for data to be received before the `recvfrom` function times out.
- For TCP sockets using TCP/IP native stack support, the receive low-water mark (the `SO_RCVLOWAT` `setsockopt` option) determines the minimum amount of data that must be received before the `recvfrom` function is completed. If the

recvfrom function times out, any data that was received is returned to the application even if the amount of data received is less than the receive low-water mark value.

- The recvfrom function cannot be issued if an activate\_on\_receipt or activate\_on\_receipt\_with\_length call is pending for the socket. These operations are mutually exclusive.

## Examples

In the following example, the application issues a recvfrom to receive a message but does not request the address of the source of the message.

```
int bytes_rcv;
int server_sock;
char data_rcv[256];
:
:
bytes_rcv = recvfrom(server_sock, data_rcv, sizeof(data_rcv), 0,
                    (struct sockaddr *) 0, (int *) 0);
```

## Related Information

- “activate\_on\_receipt — Activate a Program after Data Received” on page 144
- “getsockopt — Return Socket Options” on page 176
- “ioctl — Perform Special Operations on Socket” on page 185
- “read — Read Data on a Socket” on page 193
- “recv — Receive Data on a Connected Socket” on page 196
- “select — Monitor Read, Write, and Exception Status” on page 205
- “send — Send Data on a Connected Socket” on page 206
- “sendto — Send Data on an Unconnected Socket” on page 212
- “setsockopt — Set Options Associated with a Socket” on page 216
- “socket — Create an Endpoint for Communication” on page 223
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “write — Write Data on a Connected Socket” on page 226.

---

## recvmsg — Receive Message on Connected/Unconnected Socket

The `recvmsg` function receives messages on a socket with descriptor `s` and stores them in an array of message headers.

### Format

```
#include <socket.h>
ssize_t recvmsg(int s,
                struct msghdr *msg,
                int flags);
```

**s** The socket descriptor.

#### msg

A pointer to the message header that receives the messages.

#### flags

Must be set to 0 or one or more of the following flags. If you specify more than one flag, use the logical OR operator (`|`) to separate them:

##### MSG\_OOB

Reads any out-of-band data on the socket.

##### MSG\_PEEK

Peeks at the data that is present on the socket; the data is returned but not changed so a later receive operation sees the same data.

**Note:** Setting this parameter is supported for sockets in the `AF_INET` domain, but not supported for sockets in the `AF_IUCV` domain.

### Normal Return

If successful, the function returns the length of the data received or the entire datagram (provided the datagram fits into the specified buffer).

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** The following error codes can be returned for either TCP/IP offload support or TCP/IP native stack support.

Value	Description
<b>EINVAL</b>	<b>msg_namelen</b> is not the size of a valid address for the specified address family.
<b>EMSGSIZE</b>	Either the message is too big to be received as a single datagram or the <code>iovector</code> count is greater than or equal to <code>UIO_MAXIOV</code> , as defined in <code>socket.h</code> .
<b>ENOBUFFS</b>	Buffer space is not available to receive the message.
<b>ENOMEM</b>	There is no memory allocating buffer space to hold all messages in the <code>iovec</code> array.
<b>EWOULDBLOCK</b>	The <code>s</code> parameter is in nonblocking mode and data is not available to read.

<b>SOCBADF</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCFAULT</b>	Using <b>msg</b> would result in an attempt to access memory outside the address space of the caller..
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.

## Programming Considerations

- The `recvmsg` function receives messages on a socket with descriptor **s** and stores them in an array of message headers as defined by the `msg_hdr` structure in the `socket.h` header file.
- The `recvmsg` function applies to sockets whether they are in connected or unconnected state.
- If the data is not available for socket **s** and **s** is in blocking mode, the `recvmsg` function blocks the caller until data arrives; if **s** is in nonblocking mode, `-1` is returned and `sock_errno` is set to `EWOULDBLOCK`.
- Applications using stream sockets must place the `recvmsg` function in a loop until all data has been received.
- For sockets using TCP/IP native stack support, the receive timeout value (the `SO_RCVTIMEO` `setsockopt` option) determines how long to wait for data to be received before the `recvmsg` function times out.
- For TCP sockets using TCP/IP native stack support, the receive low-water mark (the `SO_RCVLOWAT` `setsockopt` option) determines the minimum amount of data that must be received before the `recvmsg` function ends. If the `recvmsg` function times out, any data that was received is returned to the application even if the amount of data received is less than the receive low-water mark value.
- The `recvmsg` function cannot be issued if an `activate_on_receipt` call is pending for the socket. These operations are mutually exclusive.

## Examples

In the following example, the application issues a `recvmsg` function to receive messages on a socket.

```
#include <socket.h>

struct msg_hdr      msg;
int                sock; /* UNIX socket handle */
rpc_socket_iovec_p_t iovp; /* array of bufs for rec'd data */
int                iovlen; /* number of bufs */
rpc_addr_p_t       addrp; /* address of sender */
int                *ccp; /* returned number of bytes actually rec'd */
:
:
msg.msg_iov = (struct iovec *) iovp;
msg.msg_iovlen = iovlen;
msg.msg_accrights = NULL;
msg.msg_name = (caddr_t) &(addrp->sa;
msg.msg_namelen = (addrp->len;
*(ccp) = recvmsg ((int) sock, (struct msg_hdr *) &msg, 0);
```

## Related Information

- “connect — Request a Connection to a Remote Host” on page 157
- “getsockopt — Return Socket Options” on page 176
- “ioctl — Perform Special Operations on Socket” on page 185
- “read — Read Data on a Socket” on page 193
- “recv — Receive Data on a Connected Socket” on page 196
- “recvfrom — Receive Data on Connected/Unconnected Socket” on page 199

## recvmsg

- “select — Monitor Read, Write, and Exception Status” on page 205
- “send — Send Data on a Connected Socket” on page 206
- “sendmsg — Send Message on a Socket” on page 210
- “sendto — Send Data on an Unconnected Socket” on page 212
- “setsockopt — Set Options Associated with a Socket” on page 216
- “socket — Create an Endpoint for Communication” on page 223
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “write — Write Data on a Connected Socket” on page 226
- “writev — Write Data on a Connected Socket” on page 229.

---

## 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. See the *TPF C/C++ Language Support User's Guide* for details about the `select` function.

---

## send — Send Data on a Connected Socket

The `send` function sends packets on the socket with descriptor `s`. The `send` function applies to all connected sockets.

### Format

```
#include <socket.h>
int send(int s,
         char *msg,
         int len,
         int flags);
```

**s** The socket descriptor.

#### **msg**

Pointer to the buffer containing the message to transmit.

#### **len**

Length of the message pointed to by the **msg** parameter.

When using TCP/IP offload support:

- The maximum send buffer size is 32 767 bytes when the **SO\_SNDBUF** option of the `setsockopt` function is used to increase the send buffer size.
- The default maximum size is 28 672 bytes.
- The maximum size for datagram sockets is 32 000 bytes when the **SO\_SNDBUF** option of the `setsockopt` function is used to increase the send buffer size.
- The default maximum size for datagram sockets is 9216 bytes.
- The length cannot be larger than the maximum send buffer size for this socket, which is defined by the **SO\_SNDBUF** option of the `setsockopt` function.

When using TCP/IP native stack support:

- The maximum send buffer size is 1 048 576 bytes.
- The default value of the **SO\_SNDBUF** option is 32 767.
- For a TCP socket, the maximum length that you can specify is 1 GB.
- For a UDP or RAW socket, the maximum length that you can specify is the smaller of the following values:
  - 32 KB
  - The send buffer size defined by the **SO\_SNDBUF** option.

#### **flags**

Must be set to 0 or one or more of the following flags. If you specify more than one flag, use the logical OR operator (`|`) to separate them:

##### **MSG\_OOB**

Sends out-of-band data on sockets that support it.

##### **MSG\_DONTROUTE**

The `SO_DONTROUTE` option is turned on for the duration of the operation.

### Normal Return

No indication of failure to deliver is implicit in a `send` routine. However, if it succeeds, the number of characters sent is returned.

## Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCFAULT</b>	Using <b>msg</b> and <b>len</b> results in an attempt to access a protected address space. This error code is returned only for TCP/IP offload support.
<b>SOCINVAL</b>	The value of the <b>len</b> parameter is not valid, the <b>MSG_OOB</b> option was specified for a socket that is not a stream socket, or the <b>MSG_OOB</b> option was specified, but out-of-band data is queued inline for this socket. This error code is returned only for TCP/IP native stack support.
<b>SOCNOBUFS</b>	Buffer space is not available to send the message.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCWOULDBLOCK</b>	The <b>s</b> parameter is in nonblocking mode and no buffer space is available to hold the message to be sent.
<b>SOCMSGSIZE</b>	The message was too large to be sent. This error code is returned only for TCP/IP native stack support.
<b>SOCNOTCONN</b>	The socket is not connected.
<b>EIBMIUCVERR</b>	An error occurred while the message was sent to the offload device. This error code is returned only for TCP/IP offload support.
<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.
<b>SOCTIMEDOUT</b>	The operation timed out. The socket is still available. This error code is returned only for TCP/IP native stack support.

## send

### Programming Considerations

- If buffer space is not available at the socket to hold the message to be sent, the send function normally blocks, unless the socket is in nonblocking mode. See “ioctl — Perform Special Operations on Socket” on page 185 for a description of how to set nonblocking mode.
- Use the select function to determine when to send more data.
- For sockets using TCP/IP native stack support, the send timeout value (the SO\_SNDTIMEO setsockopt option) determines how long to wait for space to become available in the send buffer before the send function times out.
- For sockets using TCP/IP native stack support:
  - For TCP sockets, if the value you specify for the len parameter is less than or equal to the send buffer size of the socket, the send process will be atomic; that is, either all of the data will be sent or none of it will be sent. If all of the data is sent, the return code is set to the value of the len parameter. If none of the data is sent, the return code is set to -1.
  - For TCP sockets, if the value you specify for the len parameter is greater than the send buffer size of the socket, the TPF system will take as much data as possible and return to the application indicating that only part of the data was processed. The application must issue more send calls for the remaining data and the application must serialize the send calls if the socket is being shared by multiple ECBs. If the send call is successful, the return code is set to a value from 1 to the value of the len parameter, which indicates how much data was sent.

### Examples

The following example sends 256 bytes. No flag is used.

```
#include <socket.h>
:
:
int bytes_sent;
int server_sock;
char data_sent[256];
:
:
bytes_sent = send(server_sock, data_sent, sizeof(data_sent), 0);
```

The following example sends 64 000 bytes. No flag is used.

```
#include <socket.h>
:
:
#define MESSAGE_SIZE 64000

int bytes_sent;
int server_sock;
int send_left;
int send_rc;
char *message_ptr;
:
:
message_ptr = malloc (MESSAGE_SIZE);
send_left = MESSAGE_SIZE;

while (send_left > 0)
{
    send_rc = send(server_sock, message_ptr, send_left, 0);
    if send_rc == -1
        break;

    send_left -= send_rc;
    message_ptr += send_rc;
} /* End While Loop */
```

## Related Information

- “connect — Request a Connection to a Remote Host” on page 157
- “getsockopt — Return Socket Options” on page 176
- “ioctl — Perform Special Operations on Socket” on page 185
- “read — Read Data on a Socket” on page 193
- “recv — Receive Data on a Connected Socket” on page 196
- “recvfrom — Receive Data on Connected/Unconnected Socket” on page 199
- “select — Monitor Read, Write, and Exception Status” on page 205
- “sendto — Send Data on an Unconnected Socket” on page 212
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223
- “write — Write Data on a Connected Socket” on page 226.

## sendmsg — Send Message on a Socket

The `sendmsg` function sends messages on a socket with descriptor `s` passed in an array of message headers.

### Format

```
#include <socket.h>
int      sendmsg(int s,
                struct msghdr *msg,
                int flags);
```

**s** The socket descriptor.

#### msg

A pointer to the message header that receives the messages.

#### flags

Must be set to 0 or one or more of the following flags. If you specify more than one flag, use the logical OR operator (`()`) to separate them:

##### MSG\_OOB

Reads any out-of-band data on the socket.

##### MSG\_DONTROUTE

The `SO_DONTROUTE` option is turned on for the duration of the operation. This is mainly used by diagnostic or routing programs.

**Note:** Setting this parameter is supported for sockets in the `AF_INET` domain, but not supported for sockets in the `AF_IUCV` domain.

### Normal Return

If successful, the function returns the length of the data sent.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** The following error codes can be returned for either TCP/IP offload support or TCP/IP native stack support.

Value	Description
<b>EINVAL</b>	<b>msg_namelen</b> is not the size of a valid address for the specified address family.
<b>EMSGSIZE</b>	Either the message is too big to be sent as a single datagram or the <code>iovector</code> count is greater than or equal to <code>UIO_MAXIOV</code> as defined in <code>socket.h</code> .
<b>ENOBUFFS</b>	Buffer space is not available to send the message.
<b>ENOMEM</b>	There is no memory allocating buffer space to hold all messages in the <code>iovec</code> array.
<b>EWOULDBLOCK</b>	The <code>s</code> parameter is in nonblocking mode and data cannot be sent.
<b>SOCBADF</b>	The <code>s</code> parameter is not a valid socket descriptor.
<b>SOCFAULT</b>	Using <b>msg</b> would result in an attempt to access memory the address space of the caller.

## SOCKET

The `s` parameter is not a valid socket descriptor.

## Programming Considerations

- The `sendmsg` function sends messages on a socket with descriptor `s` passed in an array of message headers as defined by the `msg_hdr` structure in the `socket.h` header file.
- The `sendmsg` function applies to sockets whether they are in connected or unconnected state.
- If there is not enough available buffer space to hold the socket data to be sent and socket `s` is in blocking mode, the `sendmsg` function blocks the caller until data arrives; if `s` is in nonblocking mode, `-1` is returned and `sock_errno` is set to `EWOULDBLOCK`.
- Applications using stream sockets must place the `sendmsg` function in a loop until all data has been received.

## Examples

In the following example, the application issues a `sendmsg` function to send messages on a socket.

```
#include <socket.h>
struct msg_hdr    msg;
int               sock; /* UNIX socket handle */
rpc_socket_iovec_ptr_t iovp; /* array of bufs of data to send */
int              iovlen; /* number of bufs */
rpc_addr_ptr_t   addrp; /* address of sender */
int              *ccp; /* returned number of bytes actually sent */
:
:
msg.msg_name = (caddr_t) &(addrp->sa);
msg.msg_namelen = (addrp->len);
msg.msg_iov = (struct iovec *) iovp;
msg.msg_iovlen = iovlen;
msg.msg_accrightright = NULL;
msg.msg_accrightrightlen = 0;
*(ccp) = sendmsg ((int) sock, (struct msg_hdr *) &msg, 0);
```

## Related Information

- “connect — Request a Connection to a Remote Host” on page 157
- “getsockopt — Return Socket Options” on page 176
- “ioctl — Perform Special Operations on Socket” on page 185
- “read — Read Data on a Socket” on page 193
- “recv — Receive Data on a Connected Socket” on page 196
- “recvfrom — Receive Data on Connected/Unconnected Socket” on page 199
- “recvmsg — Receive Message on Connected/Unconnected Socket” on page 202
- “select — Monitor Read, Write, and Exception Status” on page 205
- “send — Send Data on a Connected Socket” on page 206
- “sendto — Send Data on an Unconnected Socket” on page 212
- “setsockopt — Set Options Associated with a Socket” on page 216
- “socket — Create an Endpoint for Communication” on page 223
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “write — Write Data on a Connected Socket” on page 226
- “writev — Write Data on a Connected Socket” on page 229.

---

## sendto — Send Data on an Unconnected Socket

The `sendto` function sends data on unconnected sockets.

### Format

```
#include <socket.h>
int      sendto(int s,
              char *msg,
              int len,
              int flags;
              struct sockaddr *to,
              int tolen);
```

**s** The socket descriptor.

#### **msg**

Pointer to the buffer containing the message to transmit.

#### **len**

Length of the message pointed to by the **msg** parameter.

When using TCP/IP offload support:

- The maximum send buffer size is 32 767 bytes when the **SO\_SNDBUF** option of the `setsockopt` function is used to increase the send buffer size.
- The default maximum size is 28 672 bytes.
- The maximum size for datagram sockets is 32 000 bytes when the **SO\_SNDBUF** option of the `setsockopt` function is used to increase the send buffer size.
- The default maximum size for datagram sockets is 9216 bytes.
- The length cannot be larger than the maximum send buffer size for this socket, which is defined by the **SO\_SNDBUF** option of the `setsockopt` function.

When using TCP/IP native stack support:

- The maximum send buffer size is 1 048 576 bytes.
- The default value of the **SO\_SNDBUF** option is 32 767.
- For a TCP socket, the maximum length that you can specify is 1 GB.
- For a UDP or RAW socket, the maximum length that you can specify is the smaller of the following values:
  - 32 KB
  - The send buffer size defined by the **SO\_SNDBUF** option.

#### **flags**

Set to 0 the following value:

#### **MSG\_DONTRROUTE**

The **SO\_DONTRROUTE** option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

**to** Address of the target.

#### **tolen**

Size of the address pointed to by the **to** parameter.

### Normal Return

If it succeeds, `sendto` returns the number of characters sent.

## Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors. The return value of this function when used with datagram sockets does not imply failure to deliver.

**Note:** 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.

Value	Description
<b>SOCDESTADDRREQ</b>	Destination address is required.
<b>SOCFAULT</b>	Using <b>msg</b> and <b>len</b> results in an attempt to copy the address into a protected address space. This error code is returned only for TCP/IP offload support.
<b>SOCINVAL</b>	The value of the <b>tolen</b> parameter is not the size of a valid address for the specified address family.
<b>SOCMSGSIZE</b>	The message was too large to be sent. This error code is returned only for TCP/IP native stack support.
<b>SOCNOBUFS</b>	There is no buffer space available to send the message.
<b>SOCNOTCONN</b>	A stream socket was used to issue the <code>sendto</code> function, and the socket was not connected.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCWOULDBLOCK</b>	The <b>s</b> parameter is in nonblocking mode and no buffer space is available to hold the message to be sent.
<b>EIBMIUCVERR</b>	An error occurred while the message was sent to the offload device. This error code is returned only for TCP/IP offload support.
<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.
<b>SOCTIMEDOUT</b>	The operation timed out. The socket is still available. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- The `sendto` function applies to any unconnected socket.

## sendto

- Use the `send` or `write` function instead of the `sendto` function for connected sockets.
- If buffer space is not available at the socket to hold the message to be sent, the `sendto` function normally blocks, unless the socket is in nonblocking mode. See “`ioctl — Perform Special Operations on Socket`” on page 185 for a description of how to set nonblocking mode. If the socket is in nonblocking mode, `sendto` returns a `-1` and sets `sock_errno` to `SOCWOULDBLOCK`.
- For datagram sockets, this function sends the entire datagram, providing the datagram fits into the TCP/IP buffers.
- If a `bind` has not yet been issued to the socket, a `bind` is issued on behalf of the application for a non-stream socket.
- For sockets using TCP/IP native stack support, the send timeout value (the `SO_SNDTIMEO` `setsockopt` option) determines how long to wait for space to become available in the send buffer before the `sendto` function times out.
- For RAW sockets using TCP/IP native stack support, applications are required to send complete messages. Sending a message in fragments on a RAW socket is not supported.
- For sockets using TCP/IP native stack support:
  - For TCP sockets, if the value you specify for the `len` parameter is less than or equal to the send buffer size of the socket, the send process will be atomic; that is, either all of the data will be sent or none of it will be sent. If all of the data is sent, the return code is set to the value of the `len` parameter. If none of the data is sent, the return code is set to `-1`.
  - For TCP sockets, if the value you specify for the `len` parameter is greater than the send buffer size of the socket, the TPF system will take as much data as possible and return to the application indicating that only part of the data was processed. The application must issue more send calls for the remaining data and the application must serialize the send calls if the socket is being shared by multiple ECBs. If the send call is successful, the return code is set to a value from 1 to the value of the `len` parameter, which indicates how much data was sent.

## Examples

In the following example, 100 bytes are sent to an application with IP address 129.5.24.1 and port number 5001.

```
#include <socket.h>
:
int    bytes_sent;
int    server_sock;
char   send_msg[100];
struct sockaddr_in to_addr;
:
to_addr.sin_family      = AF_INET;
to_addr.sin_port       = 5001;
to_addr.sin_addr.s_addr = inet_addr("129.5.24.1");
bytes_sent = sendto(server_sock, send_msg, sizeof(send_msg), 0,
                   (struct sockaddr *)&to_addr, sizeof(to_addr));
:
:
```

## Related Information

- “`read — Read Data on a Socket`” on page 193
- “`recv — Receive Data on a Connected Socket`” on page 196
- “`recvfrom — Receive Data on Connected/Unconnected Socket`” on page 199
- “`send — Send Data on a Connected Socket`” on page 206

- “select — Monitor Read, Write, and Exception Status” on page 205
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223
- “write — Write Data on a Connected Socket” on page 226.

---

## setsockopt — Set Options Associated with a Socket

The `setsockopt` function sets options associated with a socket.

### Format

```
#include <socket.h>
int      setsockopt(int s,
                  int level,
                  int optname,
                  char *optval,
                  int optlen);
```

**s** The socket descriptor.

#### level

Level for which the option is set. Use the value **SOL\_SOCKET**.

#### optname

Name of a specified socket option. Use one of the following values:

##### SO\_BROADCAST

Toggles the ability to broadcast messages. Enabling this option lets the application send broadcast messages over **s** if the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.

##### SO\_KEEPALIVE

Toggles the ability to send probes on idle sockets to verify that the socket is still active. This option has meaning only for stream sockets. The **KEEPALIVE** option is valid only for TCP/IP native stack support when the socket sweeper is active.

##### SO\_LINGER

Waits to complete the `close` function if data is present. When this option is enabled and there is unsent data present when the `close` function is called, the calling application is blocked during the `close` function until the data is transmitted or the connection has timed out. The `close` function returns without blocking the caller. This option has meaning only for stream sockets.

##### SO\_OOBINLINE

Toggles reception of out-of-band data. Enabling this option causes out-of-band data to be placed in the normal data input queue as it is received, making it available to `recvfrom` without having to specify the **MSG\_OOB** flag in those calls. Disabling this option causes out-of-band data to be placed in the priority data input queue as it is received, making it available to `recvfrom` only by specifying the **MSG\_OOB** flag in these calls. This option has meaning only for stream sockets.

##### SO\_RCVBUF

Allows you to set the size of the receive buffer to a value to suit your application needs. The minimum size is 512 bytes. The maximum size is 1 048 576 bytes. This option has meaning only for sockets that are using TCP/IP native stack support.

##### SO\_RCVLOWAT

Allows you to set the receive buffer low-water mark, which is the minimum amount of data that must be received before a `read`, `recv`, `recvfrom`, `activate_on_receipt`, or `activate_on_receipt_with_length` function will be completed. This option has meaning only for TCP sockets that are using TCP/IP native stack support.

**SO\_RCVTIMEO**

Defines the receive timeout value, which is how long the system will wait for a read, recv, recvfrom, activate\_on\_receipt, activate\_on\_receipt\_with\_length, accept, activate\_on\_accept, or connect function to be completed before timing out the operation. A returned value of 0 indicates the system will not time out. The maximum value is 32 767 seconds. This option has meaning only for sockets that are using TCP/IP native stack support.

**SO\_REUSEADDR**

Toggles local address reuse. Enabling this option lets local addresses that are already in use to be bound. This changes the normal algorithm used in the bind function. At connect time, the system checks that no local address and port have the same remote address and port and returns error code SOCADDRINUSE if the association already exists.

**SO\_DONTRROUTE**

Toggles the routine bypass for outgoing messages. When you enable this option, outgoing messages bypass the standard routing algorithm and are directed to the appropriate network interface according to the network portion of the destination address. When enabled, this option lets you send packets only to directly connected networks (networks for which the host has an interface). This option has no meaning for stream sockets.

**SO\_SNDBUF**

Allows you to set the size of the send buffer to a value to suit your application needs. For sockets using TCP/IP native stack support, the minimum size is 512 bytes and the maximum size is 1 048 576 bytes.

**SO\_SNDLOWAT**

Allows you to set the send buffer low-water mark, which is the minimum amount of space that must be available in the send buffer to allow a select for write function to be processed. This option has meaning only for sockets that are using TCP/IP native stack support.

**SO\_SNDTIMEO**

Defines the send timeout value, which is how long the system will wait for a send, sendto, write, or writev function to be completed before timing out the operation. A returned value of 0 indicates the system will not time out. The maximum value is 32 767 seconds. This option has meaning only for sockets that are using TCP/IP native stack support.

**optval**

Pointer to option data. The **optval** and **optlen** parameters are used to pass data used by a particular command. The **optval** parameter points to a buffer containing the data needed by the command. The **optval** parameter is optional and can be set to the NULL pointer if data is not needed by the command.

**optlen**

Length of the option data. The **optlen** parameter must be set to the size of the data pointed to by **optval**.

**Normal Return**

Return code 0 indicates that the function was successful.

## setsockopt

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCADDRINUSE</b>	The address is already in use. This error code is returned only for TCP/IP offload support.
<b>SOCFAULT</b>	Using the <b>optval</b> and <b>optlen</b> parameters results in an attempt to access a protected address space. This error code is returned only for TCP/IP offload support.
<b>SOCNOPROTOOPT</b>	The <b>optname</b> parameter is not recognized, or the <b>level</b> parameter is not valid.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCINVAL</b>	The value of the <b>optlen</b> parameter is not a valid size.
<b>SOCNOBUFS</b>	There is not enough buffer space to satisfy the <code>setsockopt</code> function. This error code is returned only for TCP/IP offload support.
<b>SOCIPNOTFOUND</b>	The TPF system could not locate the header for the IP table (IPT). This error code is returned only for TCP/IP offload support.
<b>EIBMIUCVERR</b>	An error occurred while the message was sent to the offload device. This error code is returned only for TCP/IP offload support.
<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>OFFLOADTIMEOUT</b>	The offload device did not respond to the function call in the requested time. This error code is returned only for TCP/IP offload support.

### Programming Considerations

- The `setsockopt` function sets options associated with a socket.

- When specifying socket options, you must specify the name of the option and the level at which the option resides.
- For TCP/IP native stack support, the length passed must be a minimum of 4 bytes.
- When you use socket level options other than **SO\_LINGER**, **optval** points to an integer and **optlen** is set to the size of the integer. When the integer is nonzero, the option is enabled. When the integer is 0, the option is disabled. The **SO\_LINGER** option expects **optval** to point to a linger structure. This structure is defined in the following example:

```
struct linger
    int    l_onoff;           /* option on/off */
    int    l_linger;        /* linger time  */
    :
    :
```

- The `l_onoff` is set to 0 if the **SO\_LINGER** option is being disabled. A nonzero value enables the option. The `l_linger` field specifies the amount of time to wait before completing a `close` when there is still data to be sent. The units of `l_linger` are seconds.

## Examples

In the following example, out-of-band data is set in the normal input queue.

```
#include <socket.h>
:
int rc;
int server_sock;
int optval;
:
optval = 1;
rc = setsockopt(server_sock, SOL_SOCKET, SO_OOBINLINE,
               (char *)&optval, sizeof(int));
:
:
```

## Related Information

- “getsockopt — Return Socket Options” on page 176
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223.

---

## shutdown — Shut Down All or Part of a Duplex Connection

The shutdown function shuts down all or part of a duplex connection.

### Format

```
#include <socket.h>
int      shutdown(int s,
                 int how);
```

**s** The socket descriptor.

#### how

Condition of the shutdown. Use one of the following values:

- 0** No more data can be received on socket **s**.
- 1** No more data can be sent on socket **s**.
- 2** No more data can be sent or received on socket **s**.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

A return code equal to  $-1$  indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCINVAL</b>	The <b>how</b> parameter was not set to a valid value.
<b>SOCNOTCONN</b>	The socket was not connected.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>EIBMIUCVERR</b>	Error occurred when the function call was sent to the offload device. This error code is returned only for TCP/IP offload support.
<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>OFFLOADTIMEOUT</b>	The offload device did not respond to the function

call in the requested time period. This error code is returned only for TCP/IP offload support.

## Programming Considerations

None.

## Examples

In the following example, the read and write half of a duplex connection are shut down.

```
#include <socket.h>
:
int rc;
int server_sock;
:
rc = shutdown(s, 2); /*shutdown both ends */
```

## Related Information

- “accept — Accept a Connection Request” on page 138
- “close — Shut Down a Socket” on page 155
- “connect — Request a Connection to a Remote Host” on page 157
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223.

## sock\_errno — Return the Error Code Set by a Socket Call

The `sock_errno` function returns the error code set by a socket call.

### Format

```
#include <socket.h>
int      sock_errno(void);
```

### Normal Return

A nonnegative error code is returned.

### Error Return

None.

### Programming Considerations

None.

### Examples

The following example prints out the socket error number if an error occurs in the socket function.

```
#include <socket.h>
:
:
int rc;
int server_sock;
:
:
server_sock = socket(AF_INET, SOCK_STREAM, 0);
if (server_sock == -1)
{
    printf("Error in socket - %d\n", sock_errno());
    exit(0);
}
```

### Related Information

None.

## socket — Create an Endpoint for Communication

The `socket` function creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

### Format

```
#include <socket.h>
int socket(int domain,
           int type,
           int protocol);
```

#### domain

The address domain requested. Use the value **AF\_INET**. The **domain** parameter specifies a domain in which communication is to take place.

#### type

Specifies the type of socket created. The type is analogous with the semantics of the communication requested. Use one of the following values:

##### **SOCK\_STREAM**

Provides sequenced, duplex byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data.

##### **SOCK\_DGRAM**

Provides datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times.

##### **SOCK\_RAW**

Provides the interface to internal protocols (such as IP).

#### protocol

The protocol requested. Use one of the following values:

**0** Default protocol based on the **domain** and **type**.

##### **IPPROTO\_IP**

Specifies the Internet Protocol.

##### **IPPROTO\_ICMP**

Specifies the Internet Control Message protocol.

##### **IPPROTO\_TCP**

Specifies the Transmission Control Protocol (TCP). This is the default for a **type** of **SOCK\_STREAM**.

##### **IPPROTO\_UDP**

Specifies the User Datagram Protocol (UDP). This is the default for a **type** of **SOCK\_DGRAM**.

##### **IPPROTO\_RAW**

Specifies a raw IP packet.

The **protocol** parameter specifies a particular protocol to be used with the socket. In most cases, a single protocol exists to support a particular type of socket in a particular addressing family (not true with raw sockets). If the protocol field is set to 0, the system selects the default protocol number for the domain and socket type requested. Currently, protocol defaults are TCP for stream sockets and UDP for datagram sockets. There is no default for raw sockets.

## socket

### Normal Return

A nonnegative socket descriptor indicates success.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCNOBUFS</b>	There is not enough space to create a new socket because the socket block table is full. This error code is returned only for TCP/IP native stack support.
<b>SOCPROTONOSUPPORT</b>	The protocol is not supported in this domain or this protocol is not supported for this socket type.
<b>SOCPROTOTYPE</b>	The protocol is the wrong type for the socket. This error code is returned only for TCP/IP offload support.
<b>SOCAFNOSUPPORT</b>	The address family is not supported.
<b>SOC SOCKTNOSUPPORT</b>	The socket type is not supported.
<b>E1052STATE</b>	The socket was not created 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.

### Programming Considerations

When both TCP/IP native stack support and TCP/IP offload support are defined in the TPF system, the select TCP/IP support (USOK) user exit is called to select the appropriate TCP/IP support to use when a socket function is called. See *TPF System Installation Support Reference* for information about the select TCP/IP support user exit.

### Examples

In the following example, a stream socket is created.

```
#include <socket.h>
...
int server_sock;
...
server_sock = socket(AF_INET, SOCK_STREAM, 0);
```

### Related Information

- “accept — Accept a Connection Request” on page 138
- “bind — Bind a Local Name to the Socket” on page 152
- “close — Shut Down a Socket” on page 155
- “connect — Request a Connection to a Remote Host” on page 157

- “getsockname — Return the Name of the Local Socket” on page 174
- “getsockopt — Return Socket Options” on page 176
- “ioctl — Perform Special Operations on Socket” on page 185
- “recvfrom — Receive Data on Connected/Unconnected Socket” on page 199
- “select — Monitor Read, Write, and Exception Status” on page 205
- “send — Send Data on a Connected Socket” on page 206
- “sendto — Send Data on an Unconnected Socket” on page 212
- “shutdown — Shut Down All or Part of a Duplex Connection” on page 220
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “write — Write Data on a Connected Socket” on page 226
- “writev — Write Data on a Connected Socket” on page 229.

---

## write — Write Data on a Connected Socket

**Note**

This description applies only to sockets. See *TPF C/C++ Language Support User's Guide* for more information about the write function for files.

The write function writes data on a socket with descriptor **s**. The write function applies only to connected sockets.

### Format

```
#include <socket.h>
int write(int s,
          char *buf,
          int len);
```

**s** The socket descriptor.

**buf**

Pointer to the buffer holding the data to be written.

**len**

Length of the message pointed to by the **msg** parameter.

When using TCP/IP offload support:

- The maximum send buffer size is 32 767 bytes when the **SO\_SNDBUF** option of the setsockopt function is used to increase the send buffer size.
- The default maximum size is 28 672 bytes.
- The maximum size for datagram sockets is 32 000 bytes when the **SO\_SNDBUF** option of the setsockopt function is used to increase the send buffer size.
- The default maximum size for datagram sockets is 9216 bytes.
- The length cannot be larger than the maximum send buffer size for this socket, which is defined by the **SO\_SNDBUF** option of the setsockopt function.

When using TCP/IP native stack support:

- The maximum send buffer size is 1 048 576 bytes.
- The default value of the **SO\_SNDBUF** option is 32 767.
- For a TCP socket, the maximum length that you can specify is 1 GB.
- For a UDP or RAW socket, the maximum length that you can specify is the smaller of the following values:
  - 32 KB
  - The send buffer size defined by the **SO\_SNDBUF** option.

### Normal Return

If it succeeds, the function returns the number of bytes.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCFAULT</b>	Using <b>buf</b> and <b>len</b> results in an attempt to access a protected address space. This error code is returned only for TCP/IP offload support.
<b>SOCNOBUFS</b>	Buffer space is not available to send the message.
<b>SOCNOTSOCK</b>	The <b>s</b> parameter is not a valid socket descriptor.
<b>SOCINVAL</b>	An invalid length was specified.
<b>SOCWOULDBLOCK</b>	The <b>s</b> parameter is in nonblocking mode and no buffer space is available to hold the message to be sent.
<b>SOCNOTCONN</b>	The socket is not connected.
<b>EIBMIUCVERR</b>	An error occurred when the function call was sent to the offload device. This error code is returned only for TCP/IP offload support.
<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.
<b>SOCMSGSIZE</b>	The message was too large to be sent. This error code is returned only for TCP/IP native stack support.
<b>SOCTIMEDOUT</b>	The operation timed out. The socket is still available. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- This function writes up to **len** bytes of data.
- If there is no available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, `write` blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, `write` returns a `-1` and sets `sock_errno` to `SOCWOULDBLOCK`. See “`ioctl` — Perform Special Operations on Socket” on page 185 for a description of how to set the nonblocking mode.
- For sockets using TCP/IP native stack support, the send timeout value (the `SO_SNDTIMEO` `setsockopt` option) determines how long to wait for space to become available in the send buffer before the `write` function times out.
- For sockets using TCP/IP native stack support:

## write

- For TCP sockets, if the value you specify for the **len** parameter is less than or equal to the send buffer size of the socket, the send process will be atomic; that is, either all of the data will be sent or none of it will be sent. If all of the data is sent, the return code is set to the value of the **len** parameter. If none of the data is sent, the return code is set to `-1`.
- For TCP sockets, if the value you specify for the **len** parameter is greater than the send buffer size of the socket, the TPF system will take as much data as possible and return to the application indicating that only part of the data was processed. The application must issue more send calls for the remaining data and the application must serialize the send calls if the socket is being shared by multiple ECBs. If the send call is successful, the return code is set to a value from 1 to the value of the **len** parameter, which indicates how much data was sent.

## Examples

In the following example, a 100-byte message is written to socket `server_sock`.

```
#include <socket.h>
:
:
int    server_sock;
char  send_msg[100];
:
:
if (write(server_sock,send_msg,sizeof(send_msg)) < 0)
    printf("error in writing on stream socket\n");
```

## Related Information

- “connect — Request a Connection to a Remote Host” on page 157
- “getsockopt — Return Socket Options” on page 176
- “ioctl — Perform Special Operations on Socket” on page 185
- “recvfrom — Receive Data on Connected/Unconnected Socket” on page 199
- “select — Monitor Read, Write, and Exception Status” on page 205
- “send — Send Data on a Connected Socket” on page 206
- “sendto — Send Data on an Unconnected Socket” on page 212
- “setsockopt — Set Options Associated with a Socket” on page 216
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223.

## writev — Write Data on a Connected Socket

The `writev` function writes data on a socket with descriptor `s`.

### Format

```
#include <socket.h>
int writev(int s,
           struct iovec *iov,
           int iovcnt);
```

**s** The socket descriptor.

**iov**

The pointer to an array of `iovec` buffers.

**iovcnt**

Number of buffers pointed to by the `iov` parameter.

### Normal Return

If it succeeds, the function returns the number of bytes written from the buffer.

### Error Return

A return code equal to `-1` indicates an error. You can get the specific error code by calling `sock_errno`. See Appendix C, “Socket Error Return Codes” on page 365 for more information about socket errors.

**Note:** 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.

Value	Description
<b>SOCFAULT</b>	Using <code>iov</code> and <code>iovcnt</code> results in an attempt to access memory outside the caller’s address space. This error code is returned only for TCP/IP offload support.
<b>SOCNOBUFS</b>	Buffer space is not available to send the message.
<b>SOCNOTSOCK</b>	The <code>s</code> parameter is not a valid socket descriptor.
<b>SOCWOULDBLOCK</b>	The <code>s</code> parameter is in nonblocking mode and no buffer space is available to hold the message to be sent.
<b>SOCINVAL</b>	The <code>iovcnt</code> parameter was not valid or one of the fields in the <code>iov</code> array was not valid.
<b>SOCMSGSIZE</b>	The message was too large to be sent. This error code is returned only for TCP/IP native stack support.
<b>SOCNOTCONN</b>	The socket is not connected.
<b>EIBMIUCVERR</b>	An error occurred when the function call was sent to the offload device. This error code is returned only for TCP/IP offload support.
<b>E1052STATE</b>	The socket was closed because the system was in or cycling down to 1052 state.
<b>EINACT</b>	An offload device associated with the socket descriptor has been disconnected. The socket is

## writev

closed except in the following cases where the API function call must be reissued to determine if the socket is available:

- “accept — Accept a Connection Request” on page 138
- “recvfrom — Receive Data on Connected/Unconnected Socket” on page 199
- “socket — Create an Endpoint for Communication” on page 223
- “activate\_on\_receipt — Activate a Program after Data Received” on page 144.

This error code is returned only for TCP/IP offload support.

### ESYSTEMERROR

A system error has occurred and closed the socket.

### SOCTIMEDOUT

The operation timed out. The socket is still available. This error code is returned only for TCP/IP native stack support.

## Programming Considerations

- The data is gathered from the buffers specified by `iov[0]..iov[iovcnt-1]`. The `iovec` structure is called by the `writev` function.

```
struct iovec
{
    char *iov_base;
    int iov_len;
}
```

The `iovec` structure contains the following fields:

Element	Description
<code>iov_base</code>	Pointer to the buffer
<code>iov_len</code>	Length of the buffer.

The `writev` function applies only to connected sockets.

- This function writes `iov_len` bytes of data. If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, `writev` blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, `writev` returns a `-1` and sets `sock_errno` to `SOCWOULDBLOCK`. See “`ioctl — Perform Special Operations on Socket`” on page 185 for a description of how to set nonblocking mode.
- For sockets using TCP/IP native stack support, the send timeout value (the `SO_SNDTIMEO` `setsockopt` option) determines how long to wait for space to become available in the send buffer before the `writev` function times out.

## Examples

The following example sends data in two buffers of 10 bytes each.

```
#include <socket.h>
:
:
int bytes_sent;
int server_sock;
int wv_count;
char msg1[10] = "aaaaa";
```

```
char msg2[10] = "bbbb";
struct iovec wv[2];
:
:
wv[0].iov_base = msg1;
wv[1].iov_base = msg2;
wv[0].iov_len = sizeof(msg1);
wv[1].iov_len = sizeof(msg2);
wv_count = 2;
bytes_sent = writev(server_sock,wv,wv_count);
```

## Related Information

- “connect — Request a Connection to a Remote Host” on page 157
- “getsockopt — Return Socket Options” on page 176
- “ioctl — Perform Special Operations on Socket” on page 185
- “read — Read Data on a Socket” on page 193
- “recv — Receive Data on a Connected Socket” on page 196
- “recvfrom — Receive Data on Connected/Unconnected Socket” on page 199
- “select — Monitor Read, Write, and Exception Status” on page 205
- “send — Send Data on a Connected Socket” on page 206
- “sendto — Send Data on an Unconnected Socket” on page 212
- “setsockopt — Set Options Associated with a Socket” on page 216
- “sock\_errno — Return the Error Code Set by a Socket Call” on page 222
- “socket — Create an Endpoint for Communication” on page 223
- “write — Write Data on a Connected Socket” on page 226.

**writev**

---

## Part 5. Operator Procedures for Internet Server Applications

<b>Operator Procedures for the Internet Daemon</b> . . . . .	235
Internet Daemon . . . . .	235
Internet Daemon Configuration File . . . . .	235
Adding an Internet Server Application . . . . .	235
Updating the IDCF . . . . .	235
Internet Daemon Control . . . . .	236
Starting the Internet Daemon . . . . .	237
Stopping the Internet Daemon . . . . .	237
Internet Server Application . . . . .	237
Internet Server Application Control . . . . .	238
<b>Trivial File Transfer Protocol (TFTP) Server</b> . . . . .	239
Adding the Trivial File Transfer Protocol (TFTP) Server. . . . .	239
Directives for the TFTP Configuration File . . . . .	240
Creating the TFTP Configuration File . . . . .	241
Transferring and Maintaining the TFTP Configuration File . . . . .	241
<b>File Transfer Protocol (FTP) Server</b> . . . . .	243
FTP Server LOG File . . . . .	243
Adding the File Transfer Protocol (FTP) Server . . . . .	243
<b>Syslog Daemon</b> . . . . .	245
Files . . . . .	245
Syslog Daemon Configuration File . . . . .	246
Modifying the Syslog Daemon Configuration File . . . . .	249
Adding the Syslog Daemon Server . . . . .	249
Operating the Syslog Daemon . . . . .	250
Starting the Syslog Daemon . . . . .	250
Stopping the Syslog Daemon . . . . .	250
Offloading Log Files . . . . .	251
Diagnosing Syslog Daemon Configuration Problems. . . . .	251
Application Considerations . . . . .	251
<b>TPF Internet Mail Server Support</b> . . . . .	253
TPF Internet Mail Server Overview . . . . .	253
Mail Database Layout . . . . .	255
Recoup Considerations for the Mail Database . . . . .	258
TPF Internet Mail Server Configuration Files . . . . .	259
SMTP Configuration File Parameters . . . . .	259
IMAP/POP Configuration File Parameters . . . . .	262
TPF Configuration File Parameters . . . . .	263
Access List Configuration Parameters . . . . .	264
TPF Internet Mail Server Administrator or Operator Tasks. . . . .	265
Configuring the TPF System for TPF Internet Mail Server Support . . . . .	265
Adding a Domain to an Existing TPF Internet Mail Server Configuration . . . . .	267
Adding New Users to an Existing TPF Internet Mail Server Configuration . . . . .	268
Controlling the TPF Internet Mail Servers. . . . .	268
Managing Client Mailboxes . . . . .	269
TPF Internet Mail Server Client Tasks . . . . .	269



---

# Operator Procedures for the Internet Daemon

This chapter describes considerations and procedures for the Internet daemon and Internet server applications.

---

## Internet Daemon

The behavior of the Internet daemon is controlled by the Internet daemon configuration file (IDCF). Use the ZINET commands to update the IDCF and control the Internet daemon process. See *TPF Operations* for more information about the ZINET commands.

## Internet Daemon Configuration File

The IDCF contains data needed to start and control an Internet server application. The IDCF is subsystem unique and processor shared. It is stored in #IDCF1 fixed file records using 128 records.

There is an entry in the IDCF for every Internet server application that can be started by the Internet daemon. The entries are uniquely identified by the name of the Internet server application and processor ID (CPU ID). Duplicate entries (the same name and processor ID) are not allowed.

### Adding an Internet Server Application

Use the ZINET ADD command to add an Internet server application to the IDCF. Some of the parameters for this command are discussed in “Internet Server Application” on page 237.

After adding an Internet server application, enter the ZINET START command so that the Internet daemon starts an Internet daemon listener to *listen* on the specified port for your Internet server application. If you specified that listening is automatic for your Internet server application, you do not have to enter the ZINET START command for the Internet server application.

For every Internet server application that you add to the IDCF, ensure that the program specified by the PGM parameter in the ZINET ADD command is loaded.

For the WAIT, NOWAIT, and AOR process models, if you want to run the same Internet server application on multiple subsystems, you must do one of the following:

- Activate one Internet server application at a time
- Use a different Internet Protocol (IP) address or port number for each subsystem.

TCP/IP allows only one socket to be bound to a specific port number for an IP address. If you define an Internet server application with IP=ANY and you want to activate the application on multiple subsystems concurrently, you must define the Internet server application with a different port number in each subsystem.

See *TPF Operations* for more information about the ZINET ADD and ZINET START commands and for restrictions on their use.

### Updating the IDCF

Use the ZINET ALTER and ZINET DELETE commands to modify data pertaining to an Internet server application already in the IDCF. After the modifications are done, stop the Internet daemon (ZINET STOP command) and then restart the Internet daemon (ZINET START command) for the changes to take effect.

See *TPF Operations* for more information about the ZINET ALTER, ZINET DELETE, ZINET START, and ZINET STOP commands and for restrictions on their use.

## Internet Daemon Control

When the Internet daemon is started, a long-running process called the Internet daemon monitor is created. The primary functions of the Internet daemon monitor are to:

- Start Internet daemon listeners as processes for Internet server applications
- Restart an Internet daemon listener if an error causes the listener process to end.

**Note:** The Internet daemon monitor attempts to restart an Internet daemon listener a maximum of five times.

- Recycle all Internet daemon listener processes if the system activation number changes to allow an online program load (OLDR functions) to occur. Recycling includes stopping and then restarting listener processes without disrupting network traffic.

The Internet daemon listeners monitor Internet server applications, and create and monitor sockets for Internet server applications based on the process model defined for the Internet server application. For each Internet server application defined with the WAIT, NOWAIT, or AOR process model, the Internet daemon listener creates and monitors one or more sockets. For each Internet server application defined with the WAIT, NOWAIT, or DAEMON process model, the Internet daemon listener monitors the Internet server application. For each Internet server application defined with the NOLISTEN or RPC process model, the Internet daemon only starts the Internet server application; the Internet daemon listener **does not** create or monitor sockets, nor does it monitor the server application.

If an Internet server application is defined with multiple IP addresses in the IDCF, there is an Internet daemon listener process for each IP address. For an Internet server application defined with IP-ANY in the IDCF, there is only one Internet daemon listener process.

All the threshold and error controls defined in the IDCF for an Internet server application, such as the maximum number of Internet server application instances and the maximum number of times an Internet server application can end because of an error (MAXPROC and SERVERERRORS parameters, respectively, in the ZINET ADD and ZINET ALTER commands), are tracked by the individual Internet daemon listener processes. For example, if you define an Internet server application to use two IP addresses, MAXPROC-1, SERVERERRORS-3, and SERVETIME-10, then:

- Two Internet daemon listener processes are created when the Internet server application is started (because the Internet server application is defined to use two IP addresses).
- Each listener process only allows one instance of the Internet server application to run at a time (because you specified MAXPROC-1). This results in a maximum of two Internet server application processes active at any time in the TPF system.
- If an Internet server application instance ends because of an error three times in a 10-second interval (because you specified SERVERERRORS-3 and SERVETIME-10), the associated Internet daemon listener process is stopped. The other Internet daemon listener process is not affected.

See *TPF Operations* for more information about the ZINET ADD and ZINET ALTER commands.

Figure 26 shows the relationship of the Internet daemon monitor and Internet daemon listeners.

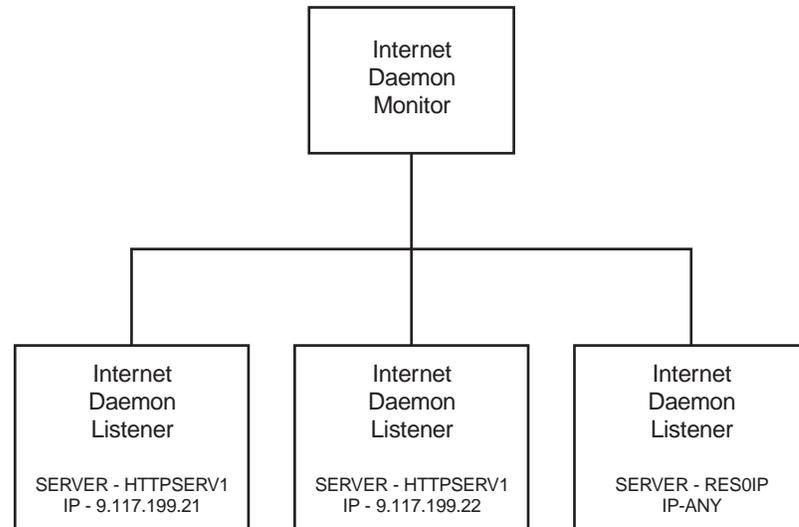


Figure 26. Relationship of the Internet Daemon Monitor and Internet Daemon Listeners. SERVER and IP are parameters in the ZINET ADD and ZINET ALTER commands.

### Starting the Internet Daemon

The Internet daemon is started in each subsystem by the cycle-up scheduler when it cycles to CRAS state or above. However, if the Internet daemon is stopped, it can be restarted by the ZINET START command. See *TPF Operations* for more information about the ZINET START command.

### Stopping the Internet Daemon

Whenever a subsystem is cycled below CRAS state, the Internet daemon is stopped. However, if the Internet daemon must be stopped, use the ZINET STOP command. See *TPF Operations* for more information about the ZINET STOP command.

---

## Internet Server Application

This section provides a discussion of some of the parameters in the ZINET ADD and ZINET ALTER commands that you must consider when adding an Internet server application or subsequently changing its characteristics. See *TPF Operations* for more information about the ZINET ADD and ZINET ALTER commands. The USER parameter must specify one of the TPF-supplied user IDs listed in the *TPF C/C++ Language Support User's Guide*.

For each Internet server application, the Internet daemon either automatically starts the listener process or the operator manually starts the Internet daemon listener. Use the ACTIVATION parameter, which can be abbreviated to ACT, to specify whether listening is automatic or manual.

- When you specify ACT-AUTO, the Internet daemon starts the listener process for the Internet server application as soon as the Internet daemon is active, unless you specify STATE-NORM.

- When you specify ACT-OPER, use the ZINET START command to start the Internet daemon listener process for an incoming message for the Internet server application. See *TPF Operations* for more information about the ZINET START command.

The Internet daemon is active starting from CRAS state, so you must decide the appropriate state in which to start your Internet server application. The STATE parameter can be specified as STATE-CRAS or STATE-NORM. Table 8 shows the interaction of the STATE and ACT parameters with the state of the TPF system.

Table 8. Relationship of STATE and ACT Parameters with the TPF System State

STATE	ACT	System State	Comment
CRAS	AUTO	CRAS or NORM	Internet server application is started automatically.
	OPER	CRAS or NORM	Internet server application can be started manually.
NORM	AUTO	CRAS	Internet server application is not started automatically and cannot be started manually.
	OPER	CRAS	Internet server application cannot be started manually.
	AUTO	NORM	Internet server application is started automatically.
	OPER	NORM	Internet server application can be started manually.

## Internet Server Application Control

Use the ZINET START command to allow the Internet daemon to start an Internet daemon listener process for the specified Internet server application. Use the ZINET STOP command to stop the Internet daemon from starting Internet daemon listener processes for the specified Internet server application. See *TPF Operations* for more information about the ZINET START and ZINET STOP commands.

**Note:** After an Internet server application is added to the IDCF, use the ZINET START command to start an Internet daemon listener process for the Internet server application.

---

## Trivial File Transfer Protocol (TFTP) Server

The behavior of the TFTP server is controlled by TFTP configuration file `/etc/tftp.conf`. This file is an EBCDIC file that is composed of directives, which are a series of interpreted commands that specify:

- The directories that the TFTP server can and cannot access in the TPF file system
- The access permissions for a file that is stored in the file system by the TFTP server on behalf of a TFTP client.

You can create and update the TFTP configuration file using one of the following:

- Your TPF system
- Another system; when you use another system, you must transfer the file to your TPF system.

There must be a TFTP configuration file for the TFTP server to process. If you create the file on another system, it must be the first file that you transfer to the TPF system using the TFTP server.

**Note:** See “Adding the Trivial File Transfer Protocol (TFTP) Server” for information about how to add the TFTP server to the Internet daemon configuration file (IDCF).

The TFTP server uses values equivalent to the following directives:

```
LOG=/tmp/tftp.log  
AUTH=444  
allow:/etc
```

---

## Adding the Trivial File Transfer Protocol (TFTP) Server

To add the TFTP server to the IDCF, enter:

```
ZINET ADD S-TFTP PGM-CTFT MODEL-model PORT-69 P-UDP IP-ipaddr  
ACT-acttype STATE-state USER-NOBODY
```

Where:

*model*

is the process model to be used by the TFTP server. Specify **WAIT** if you want only one occurrence of the TFTP server running at a time. Specify **NOWAIT** if you want multiple occurrences of the TFTP server running.

*ipaddr*

is a local intranet IP address or the value **ANY**. Typically, this type of address is assigned by a local network administrator.

**Note:** If you are using TCP/IP offload support, specify **ANY** *only* if there is just one offload device attached. If there is more than one offload device attached, you *must* specify the specific IP address for the device that you want to use.

*acttype*

is how you want to start the Internet server application. Specify **OPER** if you want to start and stop the TFTP server manually. Specify **AUTO** if you want the TFTP server to be automatically started when the Internet daemon is started.

*state*

is the lowest TPF system state in which the TFTP server can be started. Specify CRAS or NORM.

See *TPF Operations* for more information about the ZINET ADD command.

---

## Directives for the TFTP Configuration File

Each line of the TFTP configuration file must contain an EBCDIC string starting in column 1 and ending with a line-feed character (X'0A'), which is the format of a normal text file from UNIX.

**Note:** A DOS-based editor ends lines with both the carriage-return and line-feed characters (X'0D0A'). The TFTP server turns this character combination into just the line-feed character; therefore, files from either DOS or UNIX are acceptable.

The allow and deny directives control which directories and subdirectories are accessible to TFTP clients. If there are allow and deny directives for the same directory, access is denied.

# Specifies a comment line. Use comment lines sparingly because like all other lines in the configuration file, they are processed.

### AUTH

Specifies the file access permissions that are set for all files written by the TFTP server on behalf of TFTP clients using a TFTP write request. The access permissions must be three octal characters (0–7); for example, AUTH=444.

If an AUTH directive is not specified or is not specified correctly in the TFTP configuration file, access permissions are set to 444 octal. This allows owner, group, and other read access.

The access permissions for a particular file can subsequently be modified by the ZFILE chmod command. See *TPF Operations* for more information about the ZFILE chmod command.

### LOG

Specifies the name of a file in the file system where the TFTP server records status lines. If the specified file does not already exist in the TPF file system, it is created. The file name must be a fully qualified path name that begins with a slash (/).

If a LOG directive is not specified in the TFTP configuration file, the /tmp/tftp.log file is used for logging.

### allow

Specifies a directory where files can be accessed (read or write). All subdirectories are also accessible unless specifically prohibited with a deny directive. You must specify the path file name; that is, the path name must start with a slash (/).

### deny

Specifies a directory where files cannot be accessed. All subdirectories are also not accessible unless there is an allow directive for the subdirectory. You must specify the full path name; that is, the path name must start with a slash (/).

The following is an example of allow and deny directives:

```
allow:/a
deny:/a/b
```

The `/a/filename.type` and `/a/c/filename.type` files are accessible because the `allow` directive grants access and there is no `deny` directive that prevents access.

The `/filename.type` and `/e/filename.type` files are not accessible because they are not explicitly granted access by the `allow` directive. The `/a/b/filename.type` file is not accessible because access is prevented by the `deny` directive.

---

## Creating the TFTP Configuration File

The primary factor to consider when designing the TFTP configuration file is the scope of access permissions that a TFTP client can have; that is, do you want the client to be able to write in any directory and if existing files can be overwritten.

Use the following commands to create the TFTP configuration file on your TPF system:

```
zfile cat /etc/tftp.conf
zfile echo log=/tmp/tftp.log > /etc/tftp.conf
zfile echo auth:444 >> /etc/tftp.conf
zfile echo allow:/tmp >> /etc/tftp.conf
zfile cat /etc/tftp.conf
```

To create the TFTP configuration file on another system, create a file with the directives that you require and transfer the file to your TPF system. The file is automatically translated from ASCII to EBCDIC by the TFTP server when you use the ASCII or NETASCII transfer mode.

---

## Transferring and Maintaining the TFTP Configuration File

To transfer the TFTP configuration file to the TPF system, use the command that is appropriate to the system where you maintain your TFTP configuration file and specify the `to` file as `/etc/tftp.conf`.

For example, to send the TFTP configuration file to your TPF system from a personal computer (PC) Windows NT system, use the command prompt:

```
TFTP tpf.inet.addr put c.fil /etc/tftp.conf
```

Where:

*tpf.inet.addr*

is the Internet address for the TPF system or its dotted decimal equivalent.

*c.fil*

is the file name of the TFTP configuration file on your PC.

The TFTP configuration file is stored in the TPF file system with access permissions of 444 octal. This allows owner, group, and other read access. Use the ZFILE `chmod` command to change the access permissions for the TFTP configuration file. See *TPF Operations* for more information about the ZFILE `chmod` command.

To update the TFTP configuration file that already exists on your TPF system, but is maintained on another system:

1. Update the TFTP configuration file on the other system.
2. Delete the existing copy on the TPF system using the command:

```
zfile rm -f /etc/tftp.conf
```

**Note:** The `-f` parameter removes the file even though write mode is set off.

3. Transfer the updated TFTP configuration file to the TPF system as previously described.

---

## File Transfer Protocol (FTP) Server

The FTP server establishes two connections between the client and server processes; one connection for control information (commands and responses), and the other connection for the data that is transferred. The FTP server can handle both binary and text files. The files can be transferred in both directions.

The FTP client on the remote host is prompted for access information, such as the login name and password (if required), on the remote system.

The FTP server authenticates users according to the following rules:

- The login name must be in the `etc/passwd` database and cannot be a null password. A password must be provided by the client before any file operations can be performed.
- The login name must not be in the `etc/ftpusers` file.
- If the user name is `anonymous` or `ftp`, an `anonymous ftp` account must be present in the password file (the `(user ftp)`). You are allowed to log in by specifying any password (by convention, an e-mail address for the user should be used as the password).

---

### FTP Server LOG File

Each time a client logs on to FTP, a LOG file is created under the `/tmp` directory to retain FTP-related messages. Enter the `ZFILE rm` command to delete the logged records. See *TPF Operations* for more information about the `ZFILE rm` command.

---

### Adding the File Transfer Protocol (FTP) Server

To add the FTP server to the IDCF, enter:

```
ZINET ADD S-FTPD PGM-CFTP MODEL-NOWAIT PORT-21 P-TCP IP-ipaddr  
ACT-acttype STATE-state USER-ROOT
```

Where:

*ipaddr*

is a local intranet IP address or the value `ANY`. Typically, this type of address is assigned by a local network administrator.

**Note:** If you are using TCP/IP offload support, specify `ANY` **only** if there is just one offload device attached. If there is more than one offload device attached, you **must** specify the specific IP address for the device that you want to use.

*acttype*

is how you want to start the Internet server application. Specify `OPER` if you want to start and stop the FTP server manually. Specify `AUTO` if you want the FTP server to be automatically started when the Internet daemon is started.

*state*

is the lowest TPF system state in which the FTP server can be started. Specify `CRAS` or `NORM`.

See *TPF Operations* for more information about the `ZINET ADD` command.



# Syslog Daemon

The syslog daemon is a server process that provides a message logging facility for application and system processes. The syslog daemon must be started before any application program or system process that uses it starts. Internet server applications and components use the syslog daemon for logging purposes and can also send trace information to the syslog daemon. Servers on the local system use FIFO special files (also referred to as named pipes) to communicate with the syslog daemon; remote servers use TCP/IP sockets.

The syslog daemon reads and logs system messages to log files or to tape as specified by the configuration file. Remote syslog daemons can also log messages to the local syslog daemon through remote sockets. Figure 27 shows how the syslog daemon operates in the TPF environment.

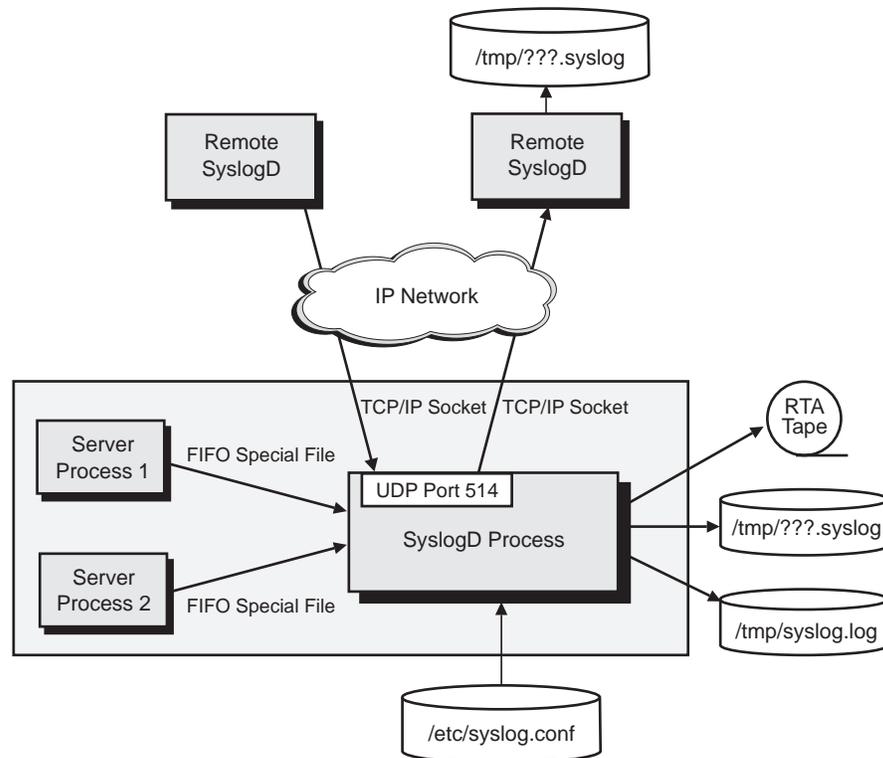


Figure 27. Syslog Daemon Operation

The syslog daemon reads the configuration file when the daemon starts and whenever the hangup signal (SIGHUP) is received. See “Syslog Daemon Configuration File” on page 246 for more information about the configuration, including the syntax for the configuration statements. See “Adding the Syslog Daemon Server” on page 249 for information about how to add the syslog daemon to the Internet daemon configuration file (IDCF).

## Files

The syslog daemon uses the following files, which reside in the basic subsystem (BSS):

File Name	Description
-----------	-------------

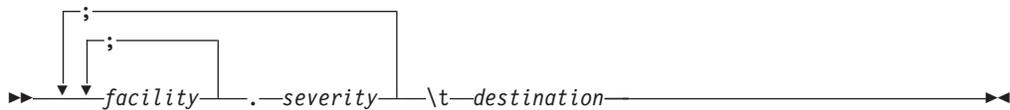
<code>/etc/syslog.pid</code>	Contains the location of the process ID. This file is created by the TPF system; <b>do not</b> modify this file.
<code>/etc/syslog.conf</code>	Default configuration file. You can modify this file as needed for your environment. See “Syslog Daemon Configuration File” for more information.

---

## Syslog Daemon Configuration File

The syslog daemon processing is controlled by a configuration file called `/etc/syslog.conf` in which you define logging rules and output destinations for error messages, authorization violation messages, and trace data. Logging rules are defined by using a *facility* name and a *severity* level. The *facility* name and *severity* level are passed on the logging request from an application when it wants to log a message. See Figure 28 on page 248 for an example of a configuration file.

Each statement of the configuration file has the following syntax:



### *facility*

is the name of the system process that is sending the message. The following facility names are supported and predefined in the syslog daemon implementation.

**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.

### **auth**

Messages generated by authorization programs.

### **daemon**

Messages generated by system server processes.

### **local0–7**

Names reserved for user-defined facilities.

### **mail**

Messages generated by a mail system.

### **news**

Messages generated by a news system.

### **syslog**

Messages generated by the syslog daemon.

### **user**

Messages generated by a process (user).

### **mark**

Messages generated by a mark signal from the syslog daemon. See “Adding the Syslog Daemon Server” on page 249 for more information about defining the syslog daemon for mark messages.

\* Placeholder used to represent all facilities.

*severity*

is the severity level of the message. The following severity levels, shown in order of importance, are supported:

**emerg**

An emergency condition; that is, the system cannot be used. This is normally broadcast to all processes.

**alert**

A condition that must be corrected immediately, such as a corrupted system database.

**crit**

A critical condition, such as a hard device error.

**err(or)**

An error message.

**warn(ing)**

A warning message.

**notice**

A condition that is not an error condition, but that may require special handling.

**info**

An informational message.

**debug**

A message that contains information normally of use only when debugging a program.

**none**

Do not log any messages for the facility.

**\t** represents the tab character.

*destination*

is the destination to which the log message will be sent. The following destinations are supported. You must use lowercase for all file names, users, and hosts.

**/file**

A specific file (for example, /tmp/syslogd/error.log). All log files used by the syslog daemon must be created in the hierarchical file system (HFS) before the syslog daemon is started.

**@host**

A syslog daemon on another host (for example, @myaixserver).

**tape**

A TPF RTA tape.

**Note:** If you direct the data to an RTA tape, you must postprocess the data offline. The data is written as null terminated strings in 4K blocks. Each block contains a header with the tape record ID of X'EA00'.

Figure 28 on page 248 shows an example of a syslog daemon configuration file. See “Modifying the Syslog Daemon Configuration File” on page 249 for information about how to modify the configuration file.

```

#
# facility.severity      destination
# -----
# Note: The facility.severity and destination must be separated by tabs.
#
# Uncomment the following to log all messages to the /dev/null file.
#*.emerg                /dev/null
#
# Uncomment the following to log all error messages (and lower)
# to the error.log file
#*.err                  /tmp/syslogd/error.log
#
# Uncomment the following to log all debug messages to tape
#*.debug                tape
#
# Uncomment the following to log all local0 informational messages (and lower)
# and local1 error messages (and lower) to a remote host
#local0.info;local1.err @remote.host.com
#
# Uncomment the following to log all daemon server debug messages
# to the server.debug file
#daemon.debug           /tmp/syslogd/server.debug
#
# Uncomment the following to log everything except local0, local1, and daemon
# messages to the garbagecan.log file
#*.emerg;local0.none;local1.none;daemon.none /tmp/syslogd/garbagecan.log

```

Figure 28. Sample /etc/syslog.conf File

#### Configuration Notes:

- Comments can be added to the configuration file by placing the hashmark (#) character in column 1 of the comment line. Everything following the hashmark character will be handled as a comment.
- When you specify a severity level, all messages with that severity **and higher** are logged at the specified destination. For example, if you specify a severity level of **error**, all messages having **error**, **crit**, **alert**, and **emerg** severities are logged. To send all messages with a severity of **error** or higher to a file named /tmp/syslogd/error.log, you can specify the following rule in the /etc/syslog.conf file:

```
*.err /tmp/syslogd/error.log
```

- You can combine logging rules and destinations in different ways. For example, to send all messages from the facility named **daemon** into one file and all messages with a severity level of **crit** or lower into another file, enter the following:

```
daemon.emerg /tmp/syslogd/daemon.log
*.crit      /tmp/syslogd/crit.log
```

**Note:** If a server sends a message to the syslog daemon with a facility name of **daemon** and a severity level of **crit**, messages will be logged in both the daemon.log and crit.log files. Likewise, if a server sends a message to the syslog daemon with a facility name of **daemon** and a severity level of **error**, the message will be logged in both files.

- If the severity level is **none**, the syslog daemon does not select any messages. For example, if you want to log all messages from facility name **local1** into one file, all messages from the daemon into another file, and all remaining messages into a third file, use the following:

```
local1.emerg /tmp/syslogd/local1.log
daemon.emerg /tmp/syslogd/daemon.log
*.emerg;local1.none;daemon.none /tmp/syslogd/the_rest.log
```

- You cannot define logging conditions related to a process name or process ID. All messages that belong to the same facility or severity class are logged in the same syslog daemon logging file whether the server task has issued the message or not.
- If the syslog daemon is running in debug mode, configuration file errors are written to the operator console because initialization is not completed until the entire configuration file has been read. See “Adding the Syslog Daemon Server” for more information about defining the syslog daemon to run in debug mode.

## Modifying the Syslog Daemon Configuration File

A default syslog daemon configuration file is provided with examples of logging rules that you can specify. You can modify this configuration file as needed for your environment. If you do not modify the configuration file, no messages will be logged (that is, all messages will be sent to the `/dev/null` file and thrown away).

To modify the syslog daemon configuration file, do the following:

1. Use TFTP or FTP to transfer the `/etc/syslog.conf` file to another system.
2. Modify the configuration file to include the logging rules that you require. You can uncomment and change the sample lines that are in the default file or just add new lines as needed.
3. Use TFTP or FTP to transfer the configuration file back to your TPF system.

---

## Adding the Syslog Daemon Server

To add the syslog daemon server to the Internet Daemon Configuration File (IDCF), enter the following from the basic subsystem (BSS):

```
ZINET ADD S-SYSLOGD PGM-CSYL MODEL-DAEMON ACT-acttype STATE-state  
USER-ROOT XPARAM-args
```

Where:

*acttype*

is how you want to start the Internet server application. Specify OPER if you want to start and stop the syslog daemon server manually. Specify AUTO if you want the syslog daemon to be started automatically when the Internet daemon is started.

*state*

is the lowest TPF system state in which the syslog daemon can be started. Specify CRAS or NORM.

*args*

is a string of parameter data that will be passed to the **argv** parameter of the `main` function defined in the specified Internet server application program. Specify one or more of the following:

**-f path**

specifies the configuration file, where *path* is the path name of the configuration file. If you do not specify this value, the default syslog daemon configuration file name of `/etc/syslog.conf` is used.

**-d** runs the syslog daemon in debugging mode. See “Diagnosing Syslog Daemon Configuration Problems” on page 251 for more information about this option.

**-m *time***

specifies a time interval that you want to log mark messages, where *time* is the time interval in minutes. A *mark message* is a time stamp labeled with MARK. These messages can be used to verify that the syslog daemon was operational during a specific time interval. If you do not specify this option, mark messages will not be logged.

**-p *pidpath***

specifies the file that contains the process ID of the syslog daemon server, where *pidpath* is the path name of the process ID file. If you do not specify this value, the default syslog daemon process ID file name of `/etc/syslog.pid` is used.

**Note:** The XPARAM parameter is optional. If you specify this parameter, it *must* be the last parameter in the command entry and you *must* specify a string of parameter data; specifying a NULL string will cause problems when starting the syslog daemon.

See *TPF Operations* for more information about the ZINET ADD command.

---

## Operating the Syslog Daemon

Use the ZINET and ZFILE commands to operate and maintain the syslog daemon.

### Starting the Syslog Daemon

If you specify AUTO for the ACTIVATION parameter when adding the syslog daemon to the IDCf, the syslog daemon will start automatically when the Internet daemon is started. If you specify OPER for the ACTIVATION parameter, you must start the syslog daemon manually. To start the syslog daemon manually, enter the following from the BSS:

#### ZINET START S-SYSLOGD

When the syslog daemon is defined to start automatically, if there is no TCP/IP transport active when the syslog daemon starts or if TCP/IP is recycled, the syslog daemon will not establish or re-establish communication with TCP/IP when it becomes available. If this occurs, you can use the syslog daemon only for local applications.

### Stopping the Syslog Daemon

To stop the syslog daemon, enter the following from the BSS:

#### ZINET STOP S-SYSLOGD

To force the syslog daemon to read its configuration file again and activate any modified parameters without stopping, enter the following to send a SIGHUP signal:

#### ZFILE kill -s SIGHUP \$(cat /etc/syslog.pid)

The syslog daemon will continue to append log messages to the files you specify in the `/etc/syslog.conf` file.

Messages are read from the FIFO special file and the Internet domain datagram socket.

## Offloading Log Files

To periodically offload log files to another location and delete unwanted messages without stopping the syslog daemon, do the following:

1. Create two syslog daemon configuration files called `/etc/syslog.conf.a` and `/etc/syslog.conf.b`. These two files will be identical except that all log files end with either a or b.
2. If your current `/etc/syslog.conf` file was created from your `syslog.conf.a` file, enter the `ZFILE cp` command to copy your `syslog.conf.b` file to the `/etc/syslog.conf` file.
3. Enter `ZFILE kill -s SIGHUP $(cat /etc/syslog.pid)` to send a SIGHUP signal to the syslog daemon. This stops the syslog daemon from writing to the a log files and forces it to write to the b log files.
4. Offload the a files using TFTP or FTP and delete their current contents.

---

## Diagnosing Syslog Daemon Configuration Problems

The syslog daemon supports a debug mode, which is selected using the `-d` argument with the XPARAM parameter of the ZINET ADD command. In this debug mode, the syslog daemon writes a large number of trace messages to the standard output (stdout) stream. You can use these messages to diagnose problems in the syslog daemon configuration or to collect documentation when reporting a syslog daemon problem to IBM support.

**Note:** Do not use the `-d` argument for normal operations.

To turn off debug mode, do one of the following:

- If you **do not** want to specify other values for the XPARAM parameter, do the following:
  1. Enter the ZINET DELETE command to delete the syslog daemon server.
  2. Enter the ZINET ADD command to add the syslog daemon server again and do not specify the XPARAM parameter.
- If you **do** want to specify other values for the XPARAM parameter, enter the ZINET ALTER command with the XPARAM parameter specified, but do not include the `-d` argument in the parameter string.

**Note:** If you specify the XPARAM parameter, you **must** provide a string of parameter data. Do not specify a NULL string for the XPARAM parameter; this will cause problems when starting the syslog daemon.

---

## Application Considerations

You can use the logging facilities of the syslog daemon server with your TPF application programs. Include the `syslog.h` header file with C programs so that these programs can open a log facility, send log messages to the syslog daemon, and close the facility:

```
#include <syslog.h>
```

```
1 openlog("tpf", LOG_PID, LOG_LOCAL0);  
2 syslog(LOG_INFO, "Hello from tpf");  
3 closelog();
```

**1** Open a log facility with a facility name of local0. Prefix each line in the log file with the program name (tpf) and the process ID.

**2** Log an informational message with the specified content.

**3** Close the log facility name.

The preceding statements created the following line in the log file:

```
<May 26 11:27:51>tpf[3014660]: Hello from tpf
```

See *TPF C/C++ Language Support User's Guide* for more information about the `syslog` function.

---

## TPF Internet Mail Server Support

TPF Internet mail server support provides a set of servers that implement the standard Internet mail protocols on the TPF system. Users, or mail clients, interact with the TPF Internet mail servers to send and retrieve Internet mail, also known as electronic mail (e-mail). The TPF system supports the following standard Internet protocols:

- Simple Mail Transfer Protocol (SMTP)
- Internet Message Access Protocol (IMAP) Version 4
- Post Office Protocol (POP) Version 3.

SMTP describes how mail messages are delivered from one computer user to another. IMAP and POP describe how mail messages that are received on a computer (that is, the mail server) are retrieved by a mail client (usually another computer, such as a workstation).

The standard Internet protocols are defined by the following Request for Comments (RFC) documents:

- RFC 821 *Simple Mail Transfer Protocol*
- RFC 2060 *Internet Message Access Protocol - Version 4rev1*
- RFC 1939 *Post Office Protocol - Version 3*
- RFC 822 *Standard for the Format of ARPA Internet Text Messages.*

For more information about these RFCs and any related extensions, go to:  
<http://www.ietf.org>

---

## TPF Internet Mail Server Overview

Figure 29 on page 254 shows the interrelationships of the actions and protocols that are involved when sending and receiving Internet mail on the TPF system.

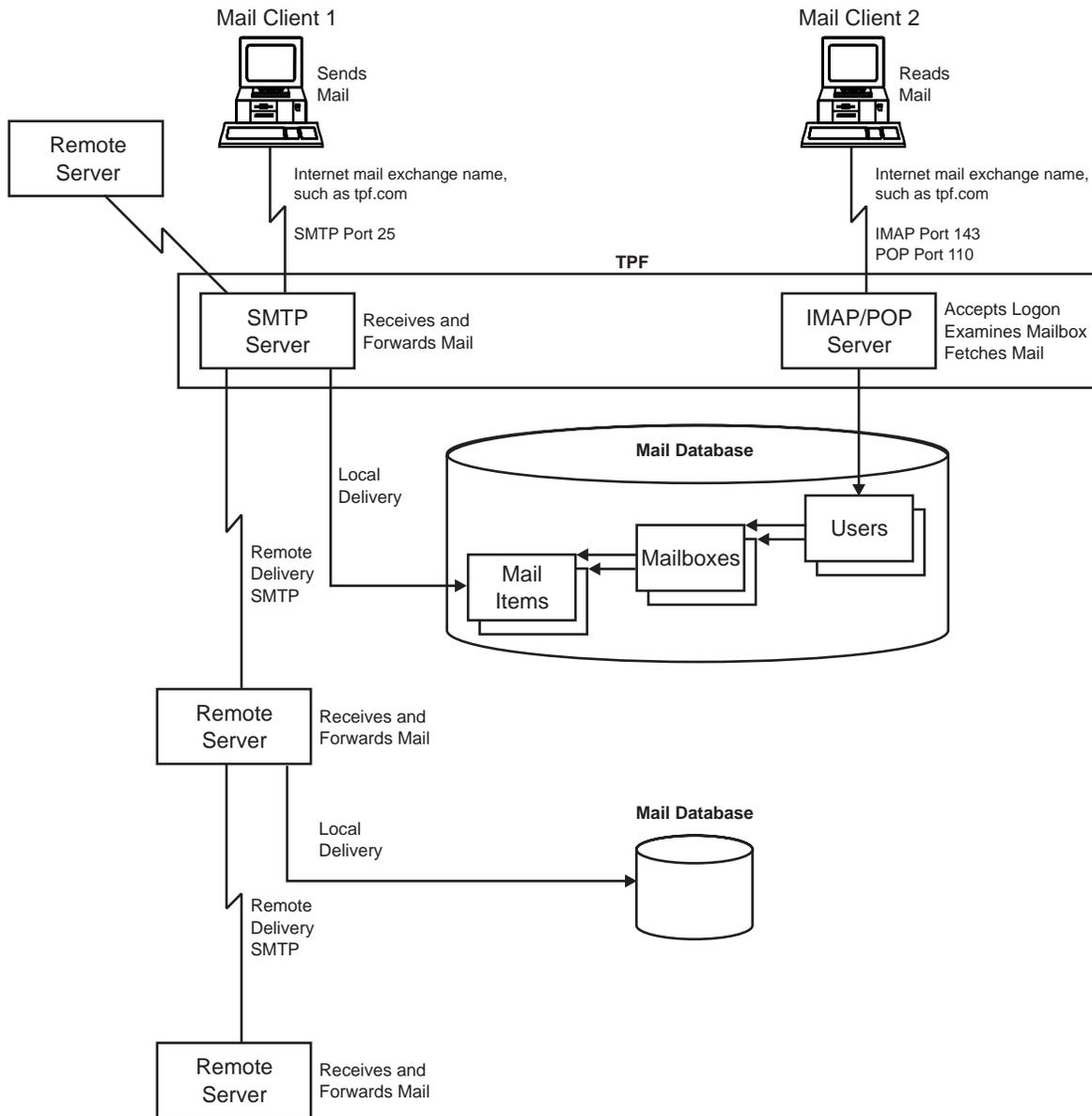


Figure 29. TPF Internet Mail Server Overview

When sending mail to another user, the mail client (Mail Client 1 in this figure) must use SMTP to put mail in the Internet mail system. Typically, the mail client uses a mail server client program such as Microsoft Outlook, Netscape, or other client program. This mail server client program connects to the SMTP server on well-known port 25 and sends the mail. On the TPF system, the SMTP server is based on the Secure Mailer, also known as Postfix. For more information about Postfix, go to: <http://www.postfix.org>.

Once the SMTP server stores the mail, the server indicates to Mail Client 1 that it has received the mail so that the client can disconnect or send more mail. The SMTP server must determine if this mail is destined for a user on a local domain (where the SMTP server is running) or for a user on a remote domain. If the mail is for a user on the local domain, the SMTP server calls the local delivery function. If

the mail is for a user on a remote domain, the SMTP server becomes an SMTP client and attempts to contact the server for that domain to transfer the mail item and delete it from local storage.

At some point, the receiving mail client (Mail Client 2) attempts to check the mail. Similarly to Mail Client 1 with SMTP, Mail Client 2 uses a mail server client program that connects to the IMAP server on well-known port 143, or to the POP server on well-known port 110, and retrieves the mail. On the TPF system, the IMAP and POP servers are based on the Cyrus project. For more information about the Cyrus project, go to: <http://www.cmu.edu/computing/cyrus>.

POP retrieves the mail item, sends the mail to the user and, optionally, deletes the copy from the mail server to free the resources of the server. IMAP allows the client to keep the mail in mailboxes on the server, and provides operations for creating, deleting, and renaming mailboxes, as well as other mail and mailbox management functions.

## Mail Database Layout

Figure 30 shows how the IMAP and POP servers access mail in mailboxes.

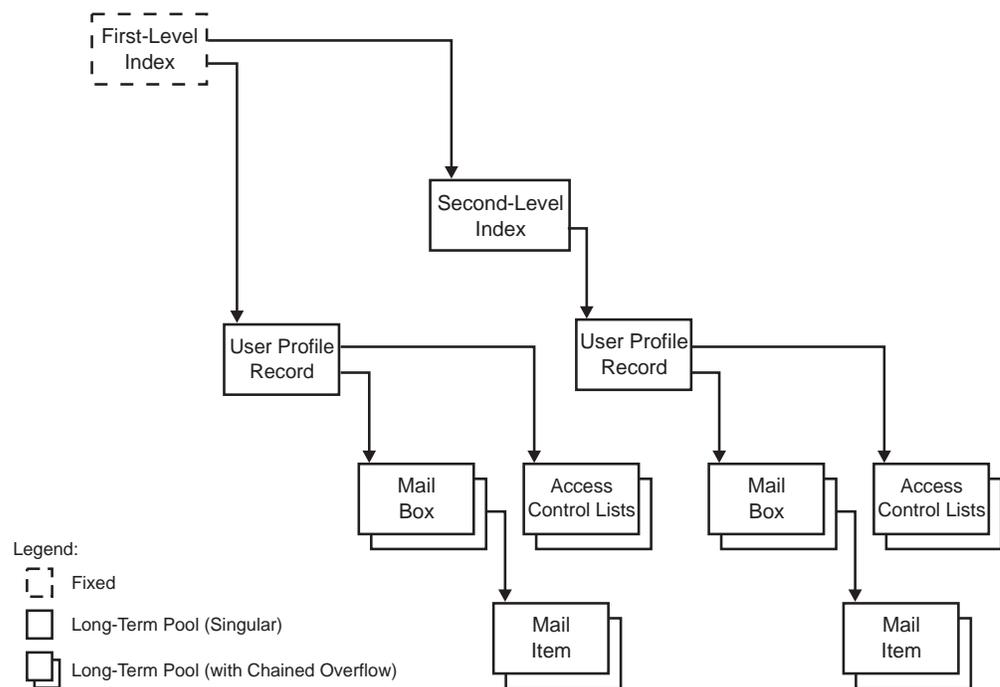


Figure 30. TPF Internet Mail Server Database — Accessing Mail

Each user has a user profile record (UPR), which is accessed through a #MAILxx fixed file record on a subsystem of your choice, where xx is a 2-character alphanumeric string. The #MAILxx record provides a first-level index to the UPR and contains pointers to either a UPR or to a second-level index record. If there are few users, the first-level index points directly to the UPRs. At some point, a second-level index is automatically created to provide better scalability for a very large number of mail clients.

There is a different #MAILxx record type associated for each domain in your mail system. For example, you can associate #MAIL01 with the mail.site1.ibm.com

domain and #MAIL02 with the mail.site2.com domain. This is defined in the TPF configuration file named /etc/tpf\_mail.conf. See "TPF Configuration File Parameters" on page 263 for more information about the TPF configuration file. Allocate 1000 #MAILxx records for each domain that you define. See *TPF System Generation* for more information about allocating fixed file records.

The UPR contains the access control information for the user and pointers to the mailbox records. Users can have more than their inbox, which is the mandatory mailbox, and can have submailboxes defined in a naming hierarchy, such as user1.inbox.projects.work.mail or user1.inbox.projects.fun.swingset. Each mailbox contains pointers to mail items; that is, messages that have been received and moved to the mailbox. All incoming mail from other users is pointed to from the inbox.

The access control information for each user is kept with the mailbox pointer in the UPR. For accountability, the access rights that the owner has given to each user is kept in a separate record called the access control list (ACL). For example, user1 can authorize user2 to read mail items in the user1.inbox.projects.fun.swingset mailbox.

Figure 31 on page 257 shows how the SMTP server accepts and delivers mail to the mailboxes.

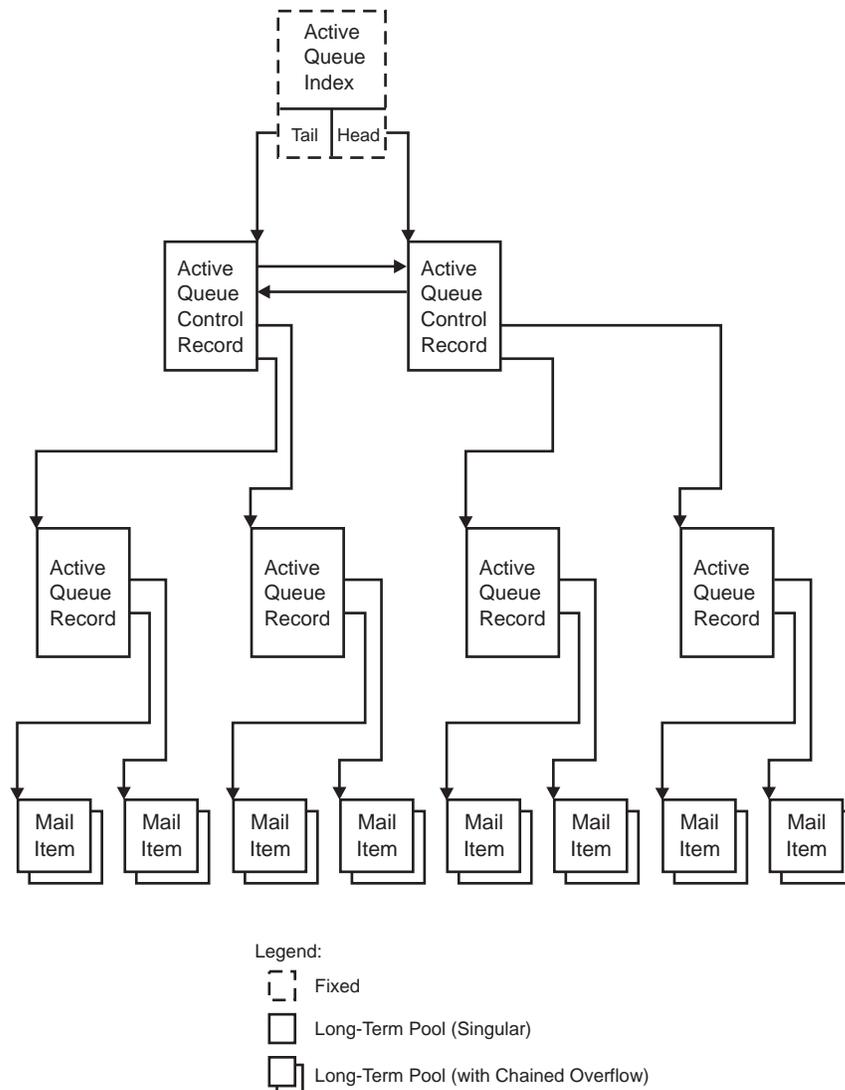


Figure 31. TPF Internet Mail Server Database — Accepting and Delivering Mail

As mail items are sent to the SMTP server, the server puts a pointer to the mail item on file in the active mail queue for delivery. There is also a deferred mail queue, similar in structure to the active mail queue, which is used to hold mail that currently cannot be delivered. Both the active and deferred mail queues are accessed through #IBMMP4 fixed file records, referred to as the active queue index record and the deferred queue index record, respectively. There is one active queue index record and one deferred queue index record for each queue. The mail queues are processor unique; therefore, if you run TPF Internet mail server support on more than one processor, mail items from a failing processor can be moved to another TPF processor in your complex and continue to be delivered.

When a new mail message is sent to the SMTP server for delivery, a pointer to the new mail item is added to an active queue record. An active queue record contains multiple pointers to mail items. When one active queue record is full, another one is created. The pointer to a new active queue record is added to an active queue control record. An active queue control record contains pointers to multiple active

queue records. As the active queue control records fill up, they are added to the end of a chain of active queue control records. Each active queue control record has a forward and backward pointer.

The active queue index record points to the head of the queue, which is the first active queue control record in the chain, and to the end of the queue, which is the last active queue control record in the chain. Mail deliveries are processed from the head of the chain, but there are additional pointers in the active queue index record that further indicate where local and remote delivery agents are currently processing. The number of these different delivery agents can be independently configured in the TPF configuration file (/etc/tpf\_mail.conf). See "TPF Configuration File Parameters" on page 263 for more information about the configuration parameters available for the TPF configuration file.

The deferred queue index record performs similarly to the active queue index record, but uses only one delivery agent because it only delivers remote mail.

---

## Recoup Considerations for the Mail Database

Before you run recoup for the TPF Internet mail server database, ensure the mail server recoup descriptor is defined correctly, and loaded on the TPF subsystem in which you plan to run the TPF Internet mail servers. Do not load the mail server recoup descriptor to subsystems in which the TPF Internet mail servers will not run.

The mail server recoup descriptor must contain a GROUP and INDEX macro pair for each #MAILxx record that you define. Segment BKD1 is initially shipped with the macro statements needed for a mail domain associated with a #MAIL01 fixed file record type, as follows:

```

SYSEQC
DC  AL2(3)      NUMBER OF PRIMARY GROUPS IN THIS CONTAINER
DC  S(IAQ,IDQ,IMAI) PRIME GROUPS
:
:
IMAI  GROUP MAC=IMAIL, ID=FC55, ECB=10, MET=(SEQ,0), NBR=1, TYP=#MAIL01, X
      TIME=900, USE=BASE, VER=0, GRP=FC50, IDCOMP=(FC66,0),      X
      ENT=INDON55, EXT=INDOFF55
      INDEX TYP=V, FI=IMAIL_LVL1_ENTY, FA=IMAIL_LVL1_ENTY+4,      X
      LI=L'IMAIL_LVL1_ENTY, CNT=(N,500), ALTID=IMAI_ALT

```

If you define additional #MAILxx fixed file record types (such as, #MAIL02, #MAIL03, and so on) or want to use a record type value other than #MAIL01, you must update segment BKD1. For example, if you want one mail domain and define the fixed file record type for that mail domain as #MAILAB instead of #MAIL01, update the value specified for the TYP parameter in the GROUP macro statement to #MAILAB, as follows:

```

SYSEQC
DC  AL2(3)      NUMBER OF PRIMARY GROUPS IN THIS CONTAINER
DC  S(IAQ,IDQ,IMAI) PRIME GROUPS
:
:
IMAI  GROUP MAC=IMAIL, ID=FC55, ECB=10, MET=(SEQ,0), NBR=1, TYP=#MAILAB, X
      TIME=900, USE=BASE, VER=0, GRP=FC50, IDCOMP=(FC66,0),      X
      ENT=INDON55, EXT=INDOFF55
      INDEX TYP=V, FI=IMAIL_LVL1_ENTY, FA=IMAIL_LVL1_ENTY+4,      X
      LI=L'IMAIL_LVL1_ENTY, CNT=(N,500), ALTID=IMAI_ALT

```

If you define more than one mail domain, you must add additional GROUP and INDEX macro statements to BKD1. For example, if you want two mail domains and define fixed file record types #MAIL01 and #MAIL02, update BKD1 as follows:

```

SYSEQC
DC    AL2(4)      NUMBER OF PRIMARY GROUPS IN THIS CONTAINER
DC    S(IAQ, IDQ, IMAI, IMAJ)  PRIME GROUPS
:
:
IMAI  GROUP MAC=IMAIL, ID=FC55, ECB=10, MET=(SEQ,0), NBR=1, TYP=#MAIL01, X
      TIME=900, USE=BASE, VER=0, GRP=FC50, IDCOMP=(FC66,0), X
      ENT=INDON55, EXT=INDOFF55
      INDEX TYP=V, FI=IMAIL_LVL1_ENTY, FA=IMAIL_LVL1_ENTY+4, X
          LI=L'IMAIL_LVL1_ENTY, CNT=(N,500), ALTID=IMAI_ALT
IMAJ  GROUP MAC=IMAIL, ID=FC55, ECB=10, MET=(SEQ,0), NBR=1, TYP=#MAIL02, X
      TIME=900, USE=BASE, VER=1, GRP=FC50, IDCOMP=(FC66,0), X
      ENT=INDON55, EXT=INDOFF55
      INDEX TYP=V, FI=IMAIL_LVL1_ENTY, FA=IMAIL_LVL1_ENTY+4, X
          LI=L'IMAIL_LVL1_ENTY, CNT=(N,500), ALTID=IMAI_ALT

```

See *TPF Database Reference* for more information about `recoup`. See *TPF System Macros* for more information about the `GROUP` and `INDEX` macros.

---

## TPF Internet Mail Server Configuration Files

The behavior of the TPF Internet mail servers is controlled by the following configuration files, which *must* be located in the basic subsystem (BSS):

- `/etc/postfix/main.cf`, which specifies the configuration parameters for the SMTP server
- `/etc/imapd.conf`, which specifies the configuration parameters for the IMAP and POP servers
- `/etc/tpf_mail.conf`, which specifies the configuration parameters for TPF-unique information.

You can also create an optional access list configuration file, named `/etc/postfix/access`. An *access list* is a file that directs the SMTP server to selectively accept or reject mail from or to specific hosts, domains, networks, host addresses, or mail addresses.

Each configuration file is an EBCDIC file that defines the configuration parameters for that server. Blank lines and lines beginning with a `#` symbol are ignored. You can create and update the configuration files by doing one of the following:

- Use the `ZFILE` commands to create and update the files directly on your TPF system.
- Create and update the files on another system and use Trivial File Transfer Protocol (TFTP) or File Transfer Protocol (FTP) to transfer the files to your TPF system.

## SMTP Configuration File Parameters

Each line of SMTP configuration file `/etc/postfix/main.cf` must be in the following format:

```
smtpparm = value
```

where *smtpparm* is the name of the configuration parameter and *value* is the parameter value that you want to specify. The following describes the configuration parameters for the `/etc/postfix/main.cf` file.

**myhostname** = *name*

Specifies the Internet host name of the TPF Internet mail system, where *name* is the host name in fully qualified domain name format. You must specify this

configuration parameter. `$myhostname`, which represents the value of the `myhostname` configuration parameter, is used as the default value in many other configuration parameters.

**mydomain = *domain***

Specifies the local Internet domain name of the TPF Internet mail system, where *domain* is the local domain name. If you do not specify this parameter, the value for `mydomain` is derived from the value defined for the `myhostname` parameter by stripping off the first component. For example, if the value of `myhostname` is defined as `tpf01.tpfmail.com`, the default value for `mydomain` is `tpfmail.com`. `$mydomain`, which represents the value of the `mydomain` configuration parameter, is used as a default value for many other configuration parameters.

**mynetworks = *ipaddr/mask***

Lists all the networks that are attached to the TPF Internet mail system, where

*ipaddr*

is an Internet Protocol (IP) address of the host in dotted decimal format.

*mask*

is a number, from 0 to 32, that represents the number of bits in the network part of the host address that are to be compared when checking the address.

For example, if you specify `172.123.255.255/8` for this parameter, any connecting client with an IP address that begins with 172 is considered local and does not require additional restriction testing. However, any connecting client whose IP address does not match 172 is considered to be remote and will continue through additional restriction testing based on other configuration parameter settings. In this example, specifying `172.0.0.0/8` for this parameter has the same effect.

You must specify this parameter. If you specify more than one network, separate each with a comma; for example:

```
mynetworks = 172.0.0.0/8, 192.255.0.0/16
```

**mail\_name = *name***

Specifies the mail system name that is used in headers of received mail, in the SMTP greeting banner, and in undeliverable (also known as bounced) mail, where *name* is any character string. If you do not specify this configuration parameter, `mail_name` is set to Postfix.

**smtpd\_banner = `$myhostname` *text***

A line of text that the TPF Internet mail server sends to the SMTP client, where *text* is any character string. You must specify `$myhostname` at the start of the text. If you do not specify this configuration parameter, `smtpd_banner` is set to `$myhostname ESMTP $mail_name`. For example, if you specify `tpf01.tpfmail.com` for the `myhostname` parameter and TPF Mail for the `mail_name` parameter, the default `smtpd_banner` will be:

```
tpf01.tpfmail.com ESMTP TPF Mail
```

**maximal\_queue\_lifetime = *numdays***

Specifies the maximum amount of time that a mail item can stay in the deferred queue before it is sent back as undeliverable, where *numdays* is the number of days. If you do not specify this parameter, a value of 5 days is used.

**minimal\_backoff\_time = *sec***

Specifies the minimum amount of time between delivery attempts of a deferred

mail item, where *sec* is the number of seconds. If you do not specify this parameter, a value of 1000 seconds is used.

**ignore\_mx\_lookup\_error = *mxerror***

Specifies how you want the delivery manager to handle errors from a mail exchange (MX) query, where *mxerror* is one of the following:

**YES**

Ignores an MX query error and proceeds to search for an address record.

**NO**

Does not ignore an MX query error and puts the mail item on the deferred queue.

If you do not specify this parameter, the value is set to NO.

**smtpd\_helo\_required = *helormsg***

Specifies whether or not the connecting client is required to send a HELO (or EHLO) command at the beginning of a session, where *helormsg* is either YES or NO. If you do not specify this parameter, the value is set to YES.

**smtpd\_helo\_restrictions = *helorstr***

Specifies the kind of restrictions that you want to apply when the client sends a HELO command, where *helorstr* is one or more of the following:

**check\_helo\_acl**

Checks the access list file (*/etc/postfix/access*) for the host name specified by the client and accepts or rejects the request based on the setting in the access list.

**permit\_mynetworks**

Allows the request when the client address matches the value of the *mynetworks* configuration parameter. If the check passes this restriction, any subsequent checks are ignored.

**reject\_invalid\_hostname**

Rejects the request if the syntax of the host name specified by the client is not valid.

Specify zero or more restrictions, separated by commas. Each restriction is applied one at a time, in the order in which you specify them. If you do not specify this configuration parameter, all of these restrictions are enabled in the following order:

- *check\_helo\_acl*
- *permit\_mynetworks*
- *reject\_invalid\_hostname*

**smtpd\_client\_restrictions = *clientrstr***

Specifies the kind of restrictions you want to apply when the client connects, where *clientrstr* is one or more of the following:

**check\_client\_acl**

Checks the access list file (*/etc/postfix/access*) for the host name of the client and accepts or rejects the request based on the setting in the access list.

**permit\_mynetworks**

Allows the request when the client address matches the value of the *mynetworks* configuration parameter. If the check passes this restriction, any subsequent checks are ignored.

**reject\_unknown\_client**

Rejects the request if the syntax of the host name of the client is not valid.

Specify zero or more restrictions, separated by commas. Each restriction is applied one at a time, in the order in which you specify them. If you do not specify this configuration parameter, all of these restrictions are enabled in the following order:

- check\_client\_acl
- permit\_mynetworks
- reject\_unknown\_client

**smtpd\_sender\_restrictions = sendrstr**

Specifies the kind of restrictions that you want to apply when the client sends a MAIL FROM command, where *sendrstr* is one or more of the following:

**check\_sender\_acl**

Checks the access list file (/etc/postfix/access) for the host name of the sending client and accepts or rejects the request based on the setting in the access list.

**reject\_unknown\_sendom**

Rejects the request if the domain portion of the host name of the sending client is not valid.

Specify zero or more restrictions, separated by commas. Each restriction is applied one at a time, in the order in which you specify them. If you do not specify this configuration parameter, all of these restrictions are enabled in the following order:

- check\_sender\_acl
- reject\_unknown\_sendom

## IMAP/POP Configuration File Parameters

Each line of IMAP/POP configuration file /etc/imapd.conf must be in the following format:

*imapparm: value*

where *imapparm* is the name of the configuration parameter and *value* is the parameter value that you want to specify.

The following lists the configuration parameters for the /etc/imapd.conf file.

**Note:** All the parameters for the IMAP/POP configuration file are optional. If you choose to use all the default settings, you **must** still create the etc/imapd.conf file. The file can be empty or it can contain one or more of the configuration parameters, but it must exist on the BSS; otherwise, the TPF Internet mail servers will not run.

**quotawarn: qstor**

Specifies the percent of storage that can be used for a mailbox or mailbox hierarchy before the TPF Internet mail server will send a warning message to the client, where *qstor* is the percent of storage. If you do not specify this configuration parameter, a value of 90 is used. Use the ZMAIL SETQUOTA command to set the storage limit for a mailbox.

**timeout: tmin**

Specifies how long the IMAP server will wait before logging off because of

inactivity, where *tmin* is the number of minutes that you want the IMAP server to wait. If you do not specify this parameter, the value is set to 30.

**poptimeout:** *popmin*

Specifies how long the POP server will wait before logging off because of inactivity, where *popmin* is the number of minutes that you want the POP server to wait. If you do not specify this parameter, the value is set to 10.

**popminpoll:** *pollmin*

Specifies the minimum amount of time that the server forces users to wait to check mail since the last time mail was checked, where *pollmin* is the number of minutes you want users to wait. If you do not specify this parameter, the value is set to 0.

**autocreatequota:** *astor*

Specifies the quota value (that is, storage limit) for a user mailbox when it is first created, where *astor* is the maximum number of 4-K records that the mailbox can use. If you do not specify this parameter, the value is set to 256. A quota value of 0 indicates that there is no storage limit. You can change this quota value by using the ZMAIL SETQUOTA command. See *TPF Operations* for more information about the ZMAIL SETQUOTA command.

## TPF Configuration File Parameters

Each line of TPF configuration file `/etc/tpf_mail.conf` must be in the following format:

*tpfparm: value*

where *tpfparm* is the name of the configuration parameter and *value* is the parameter value that you want to specify.

The following lists the configuration parameters for the `/etc/tpf_mail.conf` file.

**MAIL\_SSID:** *ss*

Specifies the name of the TPF subsystem in which you want the TPF Internet mail servers to run, where *ss* is the subsystem name. You must specify this configuration parameter.

**MAX\_LOCAL\_DELIVERY\_MANAGERS:** *maxlocal*

Specifies the maximum number of concurrent entry control blocks (ECBs) that can deliver mail to the local mailboxes, where *maxlocal* is a decimal number. You must specify this parameter. If you do not specify this parameter, the value is set to 0, which means no mail will be delivered.

**MAX\_REMOTE\_DELIVERY\_MANAGERS:** *maxremote*

Specifies the maximum number of concurrent ECBs that can deliver mail to remote mail domains, where *maxremote* is a decimal number. You must specify this parameter. If you do not specify this parameter, the value is set to 0, which means no mail will be delivered.

**MAX\_DEFERRED\_DELIVERY\_MANAGERS:** *maxdefer*

Specifies the maximum number of concurrent ECBs that can process items on the deferred mail queue, where *maxdefer* is a decimal number. You must specify this parameter. If you do not specify this parameter, the value is set to 0, which means no mail will be delivered.

**MAX\_HANGING\_RECEIVE\_MANAGERS:** *maxrcvmgr*

Specifies the maximum number of concurrent ECBs that can accept mail, where *maxrcvmgr* is a decimal number. You can use this parameter to help improve

performance. For example, if you specify a value of 50, the TPF Internet mail server will start 50 permanent mail ECBs to accept mail items and put them on the delivery queue. These ECBs will remain active unless you stop the mail server. This parameter is optional; if you do not specify this parameter, the value is set to 0. There is no maximum value, and the TPF system does not validate the number you specify.

**Note:** Each ECB requires about 40 frames; therefore, if you specify too many, the TPF system can enter input list shutdown.

**AQ\_ACK\_TIMER\_INTERVAL:** *aqacksec*

Specifies the maximum amount of time for the TPF system to acknowledge the receipt of mail items from a remote client, where *aqacksec* is the number of seconds. You must specify this parameter; the minimum value is 1.

**DQ\_ACK\_TIMER\_INTERVAL:** *dqacksec*

Specifies how often the TPF system will process mail items on the deferred mail queue, where *dqacksec* is the number of seconds. You must specify this parameter; the minimum value is 1.

**DEFER\_Q\_BACKOFF:** *qbackoffsec*

Specifies the minimum amount of time that the server will wait before attempting to deliver mail from the deferred queue after a previous attempt that failed, where *qbackoffsec* is the number of seconds. For example, assume this parameter is set to 1800 seconds. If an attempt to deliver mail failed, the server will wait at least 1800 seconds (or 30 minutes) before attempting to deliver that mail item again. You must specify this parameter; the minimum value is 0.

**MAIL\_DOMAIN<sub>xx</sub>:** *prime domain ipaddr*

Specifies the following:

*xx* The fixed file record type associated with this domain. For example, if you define the #MAIL01 fixed file record type, specify this configuration parameter as MAIL\_DOMAIN01.

*prime*

A large prime number to help distribute user accounts evenly across the database. Specify the number 249999991.

*domain*

The mail domain.

*ipaddr*

One or more Internet Protocol (IP) addresses for this domain in dotted decimal format. You can specify as many as 256 IP addresses. You **must** specify a unique value for each domain.

You can use the information in the TPF Internet mail server summary report that is generated from data collection and data reduction to tune some of the values in the TPF configuration file, such as the number of delivery managers. See *TPF System Performance and Measurement Reference* for more information about the TPF Internet mail server summary report.

## Access List Configuration Parameters

Each line of access list configuration file `/etc/postfix/access` must be in the following format:

*pattern action*

When *pattern* matches a mail address, domain, or host address, the SMTP server performs the corresponding *action*.

Specify *pattern* in one of the following ways:

*user@domain*

Matches a specific mail address.

*domain.name*

Matches the domain name itself and any subdomains of that domain, either in host names or in mail addresses. Top-level domains (such as domains that end in .com or .org) will never be matched.

*user@*

Matches all mail addresses with the specified user part.

*net.work.addr.ess*

Matches any host address in the specified network. A network address is a sequence of one or more octets separated by a period (.). You can specify all or part of a network address; for example:

- net.work.addr
- net.work
- net

Specify one of the following for *action*:

#### **ACCEPT**

Accepts mail from the address that matches the specified *pattern*.

#### **REJECT**

Rejects mail from the address that matches the specified *pattern*.

#### **DUNNO**

Continues checking other restrictions specified for this area. See the `smtpd_helo_restrictions`, `smtpd_client_restrictions` and `smtpd_sender_restrictions` parameters in the SMTP configuration file (`/etc/postfix/mail.cf`) for more information about other restrictions.

---

## **TPF Internet Mail Server Administrator or Operator Tasks**

As an administrator or operator, you can perform tasks related to:

- Controlling and configuring the mail servers
- Controlling and configuring the user accounts and user mailboxes.

## **Configuring the TPF System for TPF Internet Mail Server Support**

The following shows an example of how to configure the TPF system for TPF Internet mail server support. Consider the following scenario:

- Your mail system will have one mail domain named `flyair.tpfmail.com`, running in the basic subsystem (BSS).
  - The fixed file record type associated with domain `flyair.tpfmail.com` will be `#MAIL01`.
  - You have three employees that will be using this mail system.
  - Your mail system needs to be able to receive mail from another domain named `www.greatstuff.com`, but cannot accept mail from `www.nogood.com`.
1. Update the SIP RAMFIL macro input statements to define fixed file record type `#MAIL01` and allocate 1000 records of that type.

2. Run the FACE table generator (FCTBG) to create a new FACE table (FCTB).
3. Assemble the SIP stage I deck to create a SIP stage II deck.
4. Run SIP stage II.
5. Verify that recoup descriptor BKD1 is defined correctly for fixed file record type #MAIL01. The GROUP and INDEX macro statements will look as follows:

```

SYSEQC
DC AL2(3)      NUMBER OF PRIMARY GROUPS IN THIS CONTAINER
DC S(IAQ,IDQ,IMAI) PRIME GROUPS
:
IMAI GROUP MAC=IMAIL,ID=FC55,ECB=10,MET=(SEQ,0),NBR=1,TYP=#MAIL01, X
      TIME=900,USE=BASE,VER=0,GRP=FC50,IDCOMP=(FC66,0), X
      ENT=INDON55,EXT=INDOFF55
INDEX TYP=V,FI=IMAIL_LVL1_ENTY,FA=IMAIL_LVL1_ENTY+4, X
      LI=L'IMAIL_LVL1_ENTY,CNT=(N,500),ALTID=IMAI_ALT

```

6. IPL your TPF system and cycle the system to CRAS state or higher.
7. Enter **ZIFIL MAIL01/FC55/00/0/999/NNN/N** to initialize the #MAIL01 records.
8. Enter the following on every processor to initialize the mail-related IBMMP4 records:

```

ZIFIL IBMMP4/FC63/00/20/20/NNN/N
ZIFIL IBMMP4/FC66/00/21/21/NNN/N

```

**Note:** You only need to do this the first time you set up a mail server.

9. Enter **ZMAIL FLUSH** to initialize the database queue pointers.

**Note:** You only need to do this the first time you set up a mail server.

10. Create the SMTP configuration file (/etc/postfix/main.cf); for example:

```

#-----#
# SMTP Configuration File #
# File Name: /etc/postfix/main.cf #
#-----#
myhostname = tpf01.tpfmail.com
mydomain = tpfmail.com
mynetworks = 172.0.0.0/8, 192.0.0.0/8
mail_name = TPF Mail
smtpd_banner = $myhostname ESMTP $mail_name
maximal_queue_lifetime = 4
minimal_backoff_time = 1000
ignore_mx_lookup_error = YES
smtpd_helo_required = YES
smtpd_helo_restrictions = check_helo_acl,permit_mynetworks,reject_invalid_hostname
smtpd_client_restrictions = check_client_acl,permit_mynetworks,reject_unknown_client
smtpd_sender_restrictions = check_sender_acl,reject_unknown_sendom

```

**Note:** Ensure that the domain name is a valid, registered domain.

11. Create the IMAP/POP configuration file (/etc/imapd.conf); for example:

```

#-----#
# IMAP/POP Configuration File #
# File Name: /etc/imapd.conf #
#-----#
quotawarn: 90
timeout: 30
poptimeout: 10
popminpoll: 5
autocreatequota: 256

```

12. Create the TPF configuration file (/etc/tpf\_mail.conf); for example:

```

#-----#
# TPF Configuration File #
# File Name: /etc/tpf_mail.conf #
#-----#

```

```
MAIL_SSID: BSS
MAX_LOCAL_DELIVERY_MANAGERS: 10
MAX_REMOTE_DELIVERY_MANAGERS: 10
MAX_DEFERRED_DELIVERY_MANAGERS: 10
AQ_ACK_TIMER_INTERVAL: 30
DQ_ACK_TIMER_INTERVAL: 30
DEFER_Q_BACKOFF: 1800
MAIL_DOMAIN01: 249999991 flyair.tpfmail.com 192.168.135.55
192.168.100.200 172.234.255.88 172.234.245.20 192.221.117.23
```

13. Create the access list configuration file (/etc/postfix/access); for example:

```
#-----#
# Access List Configuration File #
# File Name: /etc/postfix/access #
#-----#
www.greatstuff.com ACCEPT
www.nogood.com REJECT
```

14. Update the external DNS servers with the domain names and IP addresses.
15. Add the SMTP, IMAP, and POP servers to the Internet daemon configuration file (IDCF) by entering the following:

```
ZINET ADD S-SMTP PGM-CMNS MODEL-DAEMON ACT-OPER
ZINET ADD S-IMAP PGM-CMNM MODEL-DAEMON ACT-OPER
ZINET ADD S-POP3 PGM-CMNP MODEL-DAEMON ACT-OPER
```

16. Enter **ZMAIL START READ** to start the IMAP and POP servers.
17. Enter **ZFILE export MAILDOMAIN=flyair.tpfmail.com** to set the MAILDOMAIN environment variable.
18. Create an account for your three users by entering the following:

```
zmail cm larry
zmail password larry larrypwd
zmail cm curly
zmail password curly curlypwd
zmail cm moe
zmail password moe moepwd
```

19. Enter **ZMAIL START ALL** to start all the TPF Internet mail servers.

See *TPF System Generation* for more information about defining and allocating fixed file records. See *TPF Operations* for more information about the ZIFIL, ZINET ADD, and ZMAIL commands.

## Adding a Domain to an Existing TPF Internet Mail Server Configuration

Assume you configured your TPF system for one mail domain, but now your business has expanded and there is a requirement to add another mail domain called trainride.tpfmail.com.

1. Update the SIP RAMFIL macro input statements to define fixed file record type #MAIL02 and allocate 1000 records of that type.
2. Run the FCTBG to create a new FACE table.
3. Assemble the SIP stage I deck to create a SIP stage II deck.
4. Run SIP stage II.
5. Update the TPF configuration file (/etc/tpf\_mail.conf); for example:

```
#-----#
# TPF Configuration File #
# File name: /etc/tpf_mail.conf #
#-----#
MAIL_SSID: BSS
```

```

MAX_LOCAL_DELIVERY_MANAGERS: 10
MAX_REMOTE_DELIVERY_MANAGERS: 10
MAX_DEFERRED_DELIVERY_MANAGERS: 10
AQ_ACK_TIMER_INTERVAL: 30
DQ_ACK_TIMER_INTERVAL: 30
DEFER_Q_BACKOFF: 1800
MAIL_DOMAIN01: 249999991 flyair.tpfmail.com 192.168.135.55
192.168.100.200 172.234.255.88 172.234.245.20 192.221.117.23
MAIL_DOMAIN02: 249999991 trainride.tpfmail.com 172.118.255.77

```

- Update recoup descriptor BKD1 to include the GROUP and INDEX macro pairs for the #MAIL02 fixed file record type, as follows:

```

SYSEQC
DC AL2(4) NUMBER OF PRIMARY GROUPS IN THIS CONTAINER
DC S(IAQ,IDQ,IMAI,IMAJ) PRIME GROUPS
:
:
IMAI GROUP MAC=IMAIL,ID=FC55,ECB=10,MET=(SEQ,0),NBR=1,TYP=#MAIL01, X
TIME=900,USE=BASE,VER=0,GRP=FC50,IDCOMP=(FC66,0), X
ENT=INDON55,EXT=INDOFF55
INDEX TYP=V,FI=IMAIL_LVL1_ENTY,FA=IMAIL_LVL1_ENTY+4, X
LI=L'IMAIL_LVL1_ENTY,CNT=(N,500),ALTID=IMAI_ALT
IMAJ GROUP MAC=IMAIL,ID=FC55,ECB=10,MET=(SEQ,0),NBR=1,TYP=#MAIL02, X
TIME=900,USE=BASE,VER=1,GRP=FC50,IDCOMP=(FC66,0), X
ENT=INDON55,EXT=INDOFF55
INDEX TYP=V,FI=IMAIL_LVL1_ENTY,FA=IMAIL_LVL1_ENTY+4, X
LI=L'IMAIL_LVL1_ENTY,CNT=(N,500),ALTID=IMAI_ALT

```

- Enter **ZMAIL STOP ALL** to stop the TPF Internet mail servers.
- IPL your TPF system and cycle to CRAS state or higher.
- Enter **ZIFIL MAIL02/FC55/00/0/999/NNN/N** to initialize the #MAIL02 records.
- Enter **ZMAIL START ALL** to restart the TPF Internet mail servers and read the updated configuration file.

See *TPF System Generation* for more information about defining and allocating fixed file records. See *TPF Operations* for more information about the ZIFIL and ZMAIL commands.

## Adding New Users to an Existing TPF Internet Mail Server Configuration

Assume you have a working TPF Internet mail system with three users. With your business continuing to expand, you have hired two new people and need to add new accounts.

- Enter the following to create the new user accounts:

```

zmail cm steveu
zmail password steveu network
zmail cm katej
zmail password katej tigger

```

See *TPF Operations* for more information about the ZMAIL commands.

## Controlling the TPF Internet Mail Servers

Use the ZMAIL command to control the TPF Internet mail servers.

The ZMAIL command allows you to:

- Start or stop the SMTP, IMAP, and POP servers
- Start or stop delivery of Internet mail to other servers

- Flush the TPF Internet mail server mail queue; that is, delete any mail that is in the active or deferred queues waiting to be delivered

## Managing Client Mailboxes

There are a number of ZMAIL commands that allow you to manage client mailboxes. With these commands you can:

- Create a new user account.
- Create or delete a mailbox.
- Display the users for a mailbox
- Display a list of mailboxes
- Display or set the storage available for a mailbox
- Change the name of a mailbox
- Set or change the access control list for a mailbox.

See *TPF Operations* for more information about the ZMAIL commands.

---

## TPF Internet Mail Server Client Tasks

As a client, you can send and receive Internet mail by doing the following:

- Configure a PC-based client, such as Netscape or Microsoft Outlook, to point to the TPF system.
- Write an application program using the `mail` function to access mail from the TPF system. See the *TPF C/C++ Language Support User's Guide* for more information about the `mail` function.

You can also control and configure submailboxes by using the ZMAIL commands.



---

## Part 6. Secure Sockets Layer (SSL) Support

<b>Secure Sockets Layer (SSL) Support</b>	273
SSL_accept	274
SSL_aor	275
SSL_check_private_key	277
SSL_connect	279
SSL_CTX_check_private_key	280
SSL_CTX_free	281
SSL_CTX_load_and_set_client_CA_list	282
SSL_CTX_load_verify_locations	283
SSL_CTX_new	285
SSL_CTX_new_shared	287
SSL_CTX_set_cipher_list	289
SSL_CTX_set_client_CA_list	292
SSL_CTX_set_default_passwd_cb_userdata	293
SSL_CTX_set_verify	294
SSL_CTX_use_certificate_chain_file	296
SSL_CTX_use_certificate_file	298
SSL_CTX_use_PrivateKey_file	300
SSL_CTX_use_RSAPrivateKey_file	302
SSL_free	304
SSL_get_cipher	305
SSL_get_error	307
SSL_get_peer_certificate	309
SSL_get_session	310
SSL_get_verify_result	311
SSL_get_version	313
SSL_library_init	314
SSL_load_and_set_client_CA_list	315
SSL_load_client_CA_file	316
SSL_new	317
SSL_pending	318
SSL_read	319
SSL_renegotiate	320
SSL_set_cipher_list	321
SSL_set_client_CA_list	324
SSL_set_fd	325
SSL_set_session	326
SSL_set_verify	327
SSL_shutdown	329
SSL_use_certificate_file	330
SSL_use_PrivateKey_file	332
SSL_use_RSAPrivateKey_file	333
SSL_write	334
SSLv2_client_method	335
SSLv2_server_method	336
SSLv23_client_method	337
SSLv23_server_method	338
SSLv3_client_method	339
SSLv3_server_method	340
TLSv1_client_method	341
TLSv1_server_method	342



---

## Secure Sockets Layer (SSL) Support

This chapter documents the Secure Sockets Layer (SSL) functions. The information in this chapter is also delivered as browser-readable hypertext markup language (HTML) files. To view this information, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Using SSL APIs** from the left navigation bar.

## SSL\_accept

---

### SSL\_accept

The SSL\_accept function accepts a Secure Sockets Layer (SSL) session connection request from a remote client application.

### Format

```
#include <openssl/ssl.h>
int SSL_accept(SSL *ssl)
```

#### ssl

A pointer to a token returned on the SSL\_new call.

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

A return code equal to 0 or a negative number indicates an error. Issue the SSL\_get\_error function to obtain specific information about the error.

### Programming Considerations

- The server application uses the SSL\_accept function to accept an SSL session from a client application. This function does not return to the application until the SSL handshake process is completed successfully or fails.
- If you are assigning many SSL sessions to the same context (CTX) structure and all of the sessions will use the same certificate, issue the SSL\_CTX\_use\_certificate\_file and SSL\_CTX\_use\_PrivateKey\_file functions once to assign the certificate to the CTX structure rather than issuing the SSL\_use\_certificate\_file once for each SSL session.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm> click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- "SSL\_connect" on page 279
- "SSL\_new" on page 317
- "SSL\_CTX\_use\_certificate\_chain\_file" on page 296
- "SSL\_CTX\_use\_certificate\_file" on page 298.

## SSL\_aor

The `SSL_aor` function allows the issuing entry control block (ECB) to exit and the specified program to be activated in a new ECB when data is available for reading on a shared Secure Sockets Layer (SSL) session.

### Format

```
#include <openssl/ssl.h>
int SSL_AOR(SSL *ssl, unsigned char *parm,
            unsigned char *pgm, unsigned int istream)
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call for the shared SSL session.

#### **parm**

A pointer to an 8-byte field. The 8 bytes of data are passed to the new ECB.

#### **pgm**

A pointer to a 4-byte field that contains the name of the TPF real-time program to activate when the new ECB is created.

#### **istream**

The I-stream number on which to activate the specified program in the new ECB. If you specify a value of 0, the least busy I-stream is selected.

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

A return code equal to 0 indicates an error. The following are the most likely causes of errors:

- The TPF real-time program specified for the **pgm** parameter does not exist.
- The value specified for the **istream** parameter is not valid.
- The value specified for the **ssl** parameter is not a token that represents a shared SSL session.

### Programming Considerations

- This function is unique to the TPF system.
- When the specified program is activated in the new ECB, it must issue an `SSL_read` call. Unlike the `socket activate_on_receipt` function that does pass data to the new ECB, this function does not.
- When the specified program is activated, the following ECB fields are set up:
 

<b>EBW004–EBW011</b>	The 8 bytes of data passed from the original ECB on the <b>parm</b> parameter.
<b>EBW012–EBW015</b>	The SSL token for this SSL session.
<b>EBW016–EBW019</b>	The socket descriptor associated with the SSL session.
<b>EBW020</b>	The timeout flag. If this is equal to 1, the <code>SSL_aor</code> function timed out and no data was received.
- The `SSL_aor` function cannot be issued if an `SSL_read` or another `SSL_aor` call is pending for this SSL session.

SSL\_aor

## Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## Related Information

- "SSL\_new" on page 317
- "SSL\_read" on page 319
- See *TPF Transmission Control Protocol/Internet Protocol* for more information about the `activate_on_receipt` function.

---

## SSL\_check\_private\_key

The `SSL_check_private_key` function verifies that the private key agrees with the corresponding public key in the certificate that is associated with the Secure Sockets Layer (SSL) structure.

### Format

```
#include <openssl/ssl.h>
int SSL_check_private_key(SSL *ssl)
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call.

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

A return code equal to 0 indicates an error. The following are the most likely causes of errors:

- The private key file does not match the corresponding public key in the certificate.
- A certificate file was not loaded.
- A key file was not loaded.

### Programming Considerations

Before you can call the `SSL_check_private_key` function, one of the following functions must be issued to set up a *private key* to the SSL session:

- `SSL_CTX_use_PrivateKey_file`
- `SSL_CTX_use_RSAPrivateKey_file`
- `SSL_use_PrivateKey_file`
- `SSL_use_RSAPrivateKey_file`.

Before you can call the `SSL_check_private_key` function, one of the following functions must be issued to set up a *certificate* to the SSL session:

- `SSL_CTX_use_certificate_file`
- `SSL_CTX_use_certificate_chain_file`
- `SSL_use_certificate_file`.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_CTX_check_private_key`” on page 280
- “`SSL_CTX_use_certificate_chain_file`” on page 296
- “`SSL_CTX_use_certificate_file`” on page 298
- “`SSL_CTX_use_PrivateKey_file`” on page 300
- “`SSL_CTX_use_RSAPrivateKey_file`” on page 302
- “`SSL_use_certificate_file`” on page 330

## SSL\_check\_private\_key

- “SSL\_use\_PrivateKey\_file” on page 332
- “SSL\_use\_RSAPrivateKey\_file” on page 333.

---

## SSL\_connect

The SSL\_connect function starts a Secure Sockets Layer (SSL) session with a remote server application.

### Format

```
#include <openssl/ssl.h>
int SSL_connect(SSL *ssl)
```

#### ssl

A pointer to a token returned on the SSL\_new call.

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

A return code equal to 0 or a negative number indicates an error. Issue the SSL\_get\_error function to obtain specific information about the error.

### Programming Considerations

- The client application uses the SSL\_connect function to start an SSL session with the server application. This function starts the SSL handshake process across the socket and does not return to the client application until the SSL handshake process is completed successfully or fails.
- If you are assigning many SSL sessions to the same context (CTX) structure and all of the sessions will use the same certificate, issue the SSL\_CTX\_use\_certificate\_file and SSL\_CTX\_use\_PrivateKey\_file functions once to assign the certificate to the CTX structure rather than issuing the SSL\_use\_certificate\_file once for each SSL session.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- "SSL\_accept" on page 274
- "SSL\_CTX\_use\_certificate\_chain\_file" on page 296
- "SSL\_CTX\_use\_certificate\_file" on page 298
- "SSL\_use\_certificate\_file" on page 330.

### SSL\_CTX\_check\_private\_key

The `SSL_CTX_check_private_key` function verifies that the private key agrees with the corresponding public key in the certificate associated with a specific context (CTX) structure.

### Format

```
#include <openssl/ssl.h>
int SSL_CTX_check_private_key(SSL_CTX *ctx)
```

#### **ctx**

A pointer to a token returned on the `SSL_CTX_new` call or the `SSL_CTX_new_shared` call.

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

A return code equal to 0 indicates an error. The following are the most likely causes of errors:

- The private key file does not match the corresponding public key in the certificate.
- A certificate file was not loaded.
- A key file was not loaded.

### Programming Considerations

You must assign a private key to the CTX structure using one of the following functions before calling the `SSL_CTX_check_private_key` function:

- `SSL_CTX_use_PrivateKey_file`
- `SSL_CTX_use_RSAPrivateKey_file`.

You must assign a certificate to the CTX structure using one of the following functions before calling the `SSL_CTX_check_private_key` function:

- `SSL_CTX_use_certificate_chain_file`
- `SSL_CTX_use_certificate_file`.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_CTX_new`” on page 285
- “`SSL_CTX_new_shared`” on page 287
- “`SSL_CTX_use_certificate_chain_file`” on page 296
- “`SSL_CTX_use_certificate_file`” on page 298
- “`SSL_CTX_use_PrivateKey_file`” on page 300
- “`SSL_CTX_use_RSAPrivateKey_file`” on page 302.

---

## SSL\_CTX\_free

The SSL\_CTX\_free function returns to the TPF system a context (CTX) structure associated with one or more SSL sessions.

### Format

```
#include <openssl/ssl.h>
void SSL_CTX_free(SSL_CTX *ctx)
```

**ctx**

A pointer to a token returned on the SSL\_CTX\_new call or the SSL\_CTX\_new\_shared call.

### Normal Return

None.

### Error Return

None.

### Programming Considerations

From the application's perspective, the CTX structure no longer exists after you issue the SSL\_CTX\_free function. However, the CTX structure is not actually returned to the TPF system until all SSL structures associated with this CTX structure have been returned.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- "SSL\_CTX\_new" on page 285
- "SSL\_CTX\_new\_shared" on page 287
- "SSL\_free" on page 304.

### SSL\_CTX\_load\_and\_set\_client\_CA\_list

The `SSL_CTX_load_and_set_client_CA_list` function loads certificates from a specific file and places the issuer name of each certificate in a specific context (CTX) structure. Each certificate is for a certificate authority (CA) that the server application trusts and is willing to accept as the CA that issued the certificate of the remote client.

#### Format

```
#include <openssl/ssl.h>
int SSL_CTX_load_and_set_client_CA_list(SSL_CTX *ctx, const char *file)
```

##### **ctx**

A pointer to a token returned on the `SSL_CTX_new` call or the `SSL_CTX_new_shared` call.

##### **file**

A pointer to the file that contains the certificates. The file must be in PEM (base64 encoded) format. The maximum length is 255 characters.

#### Normal Return

Return code 1 indicates that the function was successful.

#### Error Return

If unsuccessful, the `SSL_CTX_load_and_set_client_CA_list` function returns NULL. The following are the most likely causes of errors:

- The certificate authority (CA) file does not exist or you do not have permission to read that file.
- The CA file that contains the certificate chain is not in PEM (base64 encoded) format.

#### Programming Considerations

- This function is unique to the TPF system.
- This function combines the `SSL_load_client_CA_file` and `SSL_CTX_set_client_CA_list` functions into one function. For shared SSL sessions, you must use this function rather than the `SSL_load_client_CA_file` and `SSL_CTX_set_client_CA_list` functions.
- This function is needed only by server applications that verify the identity of remote client applications when SSL sessions are started.

#### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

#### Related Information

- “`SSL_CTX_new`” on page 285
- “`SSL_CTX_new_shared`” on page 287
- “`SSL_CTX_set_client_CA_list`” on page 292
- “`SSL_load_client_CA_file`” on page 316.

## SSL\_CTX\_load\_verify\_locations

The `SSL_CTX_load_verify_locations` function loads the certificates of the certificate authorities (CAs) that are trusted by this application and that will be used to verify certificates that are received from remote applications. Certificate revocation lists (CRLs) are also loaded if any exist.

### Format

```
#include <openssl/ssl.h>
int SSL_CTX_load_verify_locations(SSL_CTX *ctx,
                                 const char *CAfile,
                                 const char *CApath)
```

#### ctx

A pointer to a token returned on the `SSL_CTX_new` call or the `SSL_CTX_new_shared` call.

#### CAfile

A pointer to the name of the file that contains the certificates of the trusted CAs and CRLs. The file must be in PEM (base64 encoded) format. The value of this parameter can be NULL if the value of the **CApath** parameter is not NULL. The maximum length is 255 characters.

#### CApath

A pointer to the name of the directory that contains the certificates of the trusted CAs and CRLs. The files in the directory must be in PEM (base64 encoded) format. The value of this parameter can be NULL if the value of the **CAfile** parameter is not NULL. The maximum length is 255 characters.

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

A return code equal to 0 indicates an error. The following are the most likely causes of errors:

- The certificate authority (CA) file and the CA path are both NULL.
- If the CA file is not NULL, either the file does not exist or you do not have permission to read that file.
- If the CA path is not NULL, the path does not exist.

### Programming Considerations

- You must issue the `SSL_CTX_load_verify_locations` function if your application is going to verify certificates received from remote applications.
- The values of the **CAfile** and **CApath** parameters cannot both be NULL. You can specify a value of NULL for only one parameter or set both parameters to values other than NULL.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## SSL\_CTX\_load\_verify\_locations

### Related Information

- “SSL\_CTX\_new” on page 285
- “SSL\_CTX\_new\_shared” on page 287.

## SSL\_CTX\_new

The `SSL_CTX_new` function creates a new context (CTX) structure for use by one or more Secure Sockets Layer (SSL) sessions that are not shared. Use the `SSL_CTX_new_shared` function to create a CTX structure for shared SSL sessions.

### Format

```
#include <openssl/ssl.h>
SSL_CTX *SSL_CTX_new(SSL_METHOD *meth)
```

#### meth

A pointer to the connection method that indicates which SSL versions are supported and whether the new CTX structure is for a client application or a server application.

### Normal Return

Returns a pointer to the new CTX token.

### Error Return

A NULL pointer indicates an error.

## Programming Considerations

- Before calling the `SSL_CTX_new` function, you must call one of the following functions to set up the connection method:
  - `SSLv2_client_method`
  - `SSLv2_server_method`
  - `SSLv3_client_method`
  - `SSLv3_server_method`
  - `SSLv23_client_method`
  - `SSLv23_server_method`
  - `TLSv1_client_method`
  - `TLSv1_server_method`.
- Use the output of this function as input to subsequent functions that require a CTX structure as input.
- Issue the `SSL_CTX_new_shared` function to use shared SSL sessions.

## Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## Related Information

- “`SSL_CTX_free`” on page 281
- “`SSL_CTX_new_shared`” on page 287
- “`SSL_new`” on page 317
- “`SSLv2_client_method`” on page 335
- “`SSLv2_server_method`” on page 336
- “`SSLv23_client_method`” on page 337
- “`SSLv23_server_method`” on page 338

## SSL\_CTX\_new

- “SSLv3\_client\_method” on page 339
- “SSLv3\_server\_method” on page 340
- “TLSv1\_client\_method” on page 341
- “TLSv1\_server\_method” on page 342.

## SSL\_CTX\_new\_shared

The `SSL_CTX_new_shared` function creates a new context (CTX) structure for use by shared Secure Sockets Layer (SSL) sessions.

### Format

```
#include <openssl/ssl.h>
SSL_CTX *SSL_CTX_new_shared(SSL_METHOD *meth, const char *name)
```

#### meth

A pointer to the connection method that indicates which SSL versions are supported and whether the new CTX structure is for a client application or a server application.

#### name

A pointer to the name to look up in the shared SSL configuration file (`/etc/sslshared.txt`) to determine which SSL daemon processes manage the SSL sessions assigned to the CTX structure created. If you specify a NULL pointer, the TPF system selects an SSL daemon process that has the least number of SSL sessions to manage the sessions for this CTX structure.

### Normal Return

Returns a pointer to a CTX token.

### Error Return

A NULL pointer indicates an error. The most likely cause of this error is that a **name** parameter that does not exist in the shared SSL configuration file (`/etc/sslshared.txt`) was passed in.

## Programming Considerations

- Before calling the `SSL_CTX_new` function, you must call one of the following functions to set up the connection method:
  - `SSLv2_client_method`
  - `SSLv2_server_method`
  - `SSLv3_client_method`
  - `SSLv3_server_method`
  - `SSLv23_client_method`
  - `SSLv23_server_method`
  - `TLSv1_client_method`
  - `TLSv1_server_method`.
- This function is unique to the TPF system.
- Use this function rather than the `SSL_CTX_new` function when the application wants its SSL sessions to be shared. Any SSL sessions assigned to the CTX structure created by this function will be shared SSL sessions.
- Use the output of this function as the input to subsequent SSL function calls that require a pointer to a CTX structure.

## Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## SSL\_CTX\_new\_shared

### Related Information

- “SSL\_CTX\_new” on page 285
- “SSLv2\_client\_method” on page 335
- “SSLv2\_server\_method” on page 336
- “SSLv23\_client\_method” on page 337
- “SSLv23\_server\_method” on page 338
- “SSLv3\_client\_method” on page 339
- “SSLv3\_server\_method” on page 340
- “TLSv1\_client\_method” on page 341
- “TLSv1\_server\_method” on page 342.

## SSL\_CTX\_set\_cipher\_list

The `SSL_CTX_set_cipher_list` function sets ciphers for use by Secure Sockets Layer (SSL) sessions that are started using the specified context (CTX) structure. A CTX structure is needed for each application that is running SSL. Each SSL session has an SSL structure that points to a CTX structure.

### Format

```
#include <openssl/ssl.h>
int SSL_CTX_set_cipher_list(SSL_CTX *ctx, const char *str)
```

#### ctx

A pointer to a token returned on the `SSL_CTX_new` call or the `SSL_CTX_new_shared` call.

#### str

A pointer to a string that contains one or more ciphers separated by a colon, comma, or blank. The maximum length is 255 characters.

You must specify the ciphers in order of preference from highest to lowest. The TPF system supports the following SSL version 3 and Transport Layer Security (TLS) version 1 ciphers for use by the SSL sessions established from this CTX structure. The Rivest-Shamir-Adelman (RSA) key exchange is used.

<b>NULL-MD5</b>	No data encryption; MD5 for message integrity.
<b>NULL-SHA</b>	No data encryption; SHA for message integrity.
<b>EXP-RC4-MD5</b>	Export RC4 (40-bit key) for data encryption; MD5 for message integrity.
<b>RC4-MD5</b>	RC4 (128-bit key) for data encryption; MD5 for message integrity.
<b>RC4-SHA</b>	RC4 (128-bit key) for data encryption; SHA for message integrity.
<b>EXP-RC2-CBC-MD5</b>	Export RC2 (40-bit key) for data encryption; MD5 for message integrity.
<b>EXP-DES-CBC-SHA</b>	Export DES (40-bit key) for data encryption; SHA for message integrity.
<b>DES-CBC-SHA</b>	DES (56-bit key) for data encryption; SHA for message integrity.
<b>DES-CBC3-SHA</b>	Triple-DES (168-bit key) for data encryption; SHA for message integrity.

The TPF system supports the following SSL version 2 ciphers for use by the SSL sessions established from this CTX structure. The RSA key exchange is used.

<b>RC4-MD5</b>	RC4 (128-bit key) for data encryption; MD5 for message integrity.
<b>EXP-RC4-MD5</b>	Export RC4 (40-bit key) for data encryption; MD5 for message integrity.
<b>RC2-CBC-MD5</b>	RC2 (128-bit key) for data encryption; MD5 for message integrity.
<b>EXP-RC2-CBC-MD5</b>	Export RC2 (40-bit key) for data encryption; MD5 for message integrity.

## SSL\_CTX\_set\_cipher\_list

### DES-CBC-MD5

DES (56-bit key) for data encryption; MD5 for message integrity.

### DES-CBC3-MD5

Triple-DES(168-bit key) for data encryption; MD5 for message integrity.

## Normal Return

Return code 1 indicates that the function was successful.

## Error Return

A return code equal to 0 indicates an error.

## Programming Considerations

- When an SSL structure is first created using the `SSL_new` function, the structure inherits the cipher list assigned to the context (CTX) structure that was used to create the SSL structure. The `SSL_set_cipher_list` function overrides that cipher list for a specific SSL structure.
- If you are assigning many SSL sessions to the same CTX structure and each session will use the same cipher list, issue the `SSL_CTX_set_cipher_list` function once to assign the cipher list to the CTX structure rather than issuing the `SSL_set_cipher_list` function once for each SSL session.
- If you start an SSL session without issuing the `SSL_CTX_set_cipher_list` or the `SSL_set_cipher_list` functions, the system default cipher list is used.

The default ciphers for SSL version 2 are:

- DES-CBC-MD5
- DES-CBC3-MD5
- EXP-RC2-CBC-MD5
- EXP-RC4-MD5
- RC2-CBC-MD5
- RC4-MD5.

The default ciphers for SSL version 3 are:

- DES-CBC-SHA
- DES-CBC3-SHA
- EXP-DES-CBC-SHA
- EXP-RC2-CBC-MD5
- EXP-RC4-MD5
- RC4-SHA.

The default ciphers for TLS version 1 are:

- DES-CBC3-SHA
- DES-CBC-SHA
- EXP-DES-CBC-SHA
- EXP-RC2-CBC-MD5
- EXP-RC4-MD5
- RC4-MD5
- RC4-SHA.

## Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## Related Information

- "SSL\_CTX\_new\_shared" on page 287
- "SSL\_new" on page 317
- "SSL\_set\_cipher\_list" on page 321.

### SSL\_CTX\_set\_client\_CA\_list

The `SSL_CTX_set_client_CA_list` function identifies the list of certificate authorities (CAs) that will be sent to the remote client application when requesting the client certificate for any Secure Sockets Layer (SSL) session associated with a specific context (CTX) structure. The client application must provide a certificate that was signed by one of the CAs in the list.

### Format

```
#include <openssl/ssl.h>
void SSL_CTX_set_client_CA_list(SSL_CTX *ctx,STACK_OF(X509_NAME) *list)
```

#### **ctx**

A pointer to a token returned on the `SSL_CTX_new` call.

#### **list**

A pointer to a stack of CA names.

### Normal Return

None.

### Error Return

None.

### Programming Considerations

- The `SSL_CTX_set_client_CA_list` function is only needed by server applications that verify the identity of remote client applications when SSL sessions are started.
- Issue the `SSL_load_client_CA_file` function to create the list of CA names that are passed to the `SSL_CTX_set_client_CA_list` function.
- If the `SSL_CTX_set_client_CA_list` function is not used and you request a client certificate, the list of CA names that get passed to the client application are the CAs from the `SSL_CTX_load_verify_locations` function.
- You cannot use the `SSL_CTX_set_client_CA_list` function for shared SSL sessions. You must use the `SSL_CTX_load_and_set_client_CA_list` function.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_CTX_load_and_set_client_CA_list`” on page 282
- “`SSL_CTX_load_verify_locations`” on page 283
- “`SSL_CTX_new_shared`” on page 287
- “`SSL_load_client_CA_file`” on page 316.

---

## SSL\_CTX\_set\_default\_passwd\_cb\_userdata

The `SSL_CTX_set_default_passwd_cb_userdata` function identifies the password that is used to access data in a private key file that is in PEM (base64 encoded) format.

### Format

```
#include <openssl/ssl.h>
void SSL_CTX_set_default_passwd_cb_userdata(SSL_CTX *ctx, void *password)
```

**ctx**

A pointer to a token returned on the `SSL_CTX_new` call or the `SSL_CTX_new_shared` call.

**password**

A pointer to the password. The maximum length is 255 characters.

### Normal Return

None.

### Error Return

None.

### Programming Considerations

You must use the `SSL_CTX_set_default_passwd_cb_userdata` function if your application is going to load a private key file that contains encrypted data.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_CTX_new_shared`” on page 287
- “`SSL_CTX_use_PrivateKey_file`” on page 300
- “`SSL_CTX_use_RSAPrivateKey_file`” on page 302
- “`SSL_use_PrivateKey_file`” on page 332
- “`SSL_use_RSAPrivateKey_file`” on page 333.

---

## SSL\_CTX\_set\_verify

The `SSL_CTX_set_verify` function indicates whether or not to verify the identity of remote peers starting Secure Sockets Layer (SSL) sessions associated with a specific context (CTX) structure.

### Format

```
#include <openssl/ssl.h>
void SSL_CTX_set_verify(SSL_CTX *ctx, int mode, int (*cb)
                       (int, X509_STORE_CTX*))
```

#### ctx

A pointer to a token returned on the `SSL_CTX_new` call or the `SSL_CTX_new_shared` call.

#### mode

One or more of the following verify options:

##### SSL\_VERIFY\_NONE

Use this option if you do not want to verify the identity of the remote peer. This option must be used alone; no other options can be specified. Consider the following when using this option:

- If the application is a server, the application will not request the certificate for the remote client application when the SSL session is started.
- If the application is a client, the certificate for the remote server application will be validated; however, the SSL session will be started regardless of whether or not the certificate for the remote server application is valid. Issue the `SSL_get_verify_result` function to check whether or not the certificate for the server application is valid.

##### SSL\_VERIFY\_PEER

Use this option to verify the identify of the remote peer when the SSL session is started. Consider the following when using this option:

- If the application is a server, the application will request and verify the certificate for the remote client application when the SSL session is started. If the remote client application provides a certificate that is not valid, the SSL session fails.
- If the application is a client, the certificate for the remote server application is validated. If the certificate for the remote server application is not valid, the SSL session fails.

##### SSL\_VERIFY\_FAIL\_IF\_NO\_PEER\_CERT

Use this option to request that the remote client application send its certificate when the SSL session is starting, and to end the SSL session if no certificate is provided. This option only has meaning if the `SSL_VERIFY_PEER` option is also set. Do not use this option when your application is the client.

##### SSL\_VERIFY\_CLIENT\_ONCE

Use this option to verify the identity of the remote client application when the SSL session is first started. If the SSL session is renegotiated, do not verify the identify of the client application again. This option only has meaning if the `SSL_VERIFY_PEER` option is also set. Do not use this option when your application is the client.

**cb** A pointer set to `NULL`.

## Normal Return

None.

## Error Return

None.

## Programming Considerations

The default value for the verify mode is `SSL_VERIFY_NONE` when the `CTX` structure is created. You must use the `SSL_CTX_set_verify` function or the `SSL_set_verify` function with the `SSL_VERIFY_PEER` option if you want to authenticate remote peers when SSL sessions are started.

## Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## Related Information

- “`SSL_CTX_new`” on page 285
- “`SSL_CTX_new_shared`” on page 287
- “`SSL_get_verify_result`” on page 311
- “`SSL_set_verify`” on page 327.

### SSL\_CTX\_use\_certificate\_chain\_file

The `SSL_CTX_use_certificate_chain_file` function loads the chain of certificates for use with a Secure Sockets Layer (SSL) session using a specific context (CTX) structure.

#### Format

```
#include <openssl/ssl.h>
int SSL_CTX_use_certificate_chain_file(SSL_CTX *ctx, const char *file)
```

##### ctx

A pointer to a token returned on the `SSL_CTX_new` call or the `SSL_CTX_new_shared` call.

##### file

A pointer to the name of the file that contains the chain of certificates. The file that contains the certificate chain must be in PEM (base64 encoded) format. The maximum length is 255 characters.

#### Normal Return

Return code 1 indicates that the function was successful.

#### Error Return

A return code equal to 0 indicates an error. The following are the most likely causes of errors:

- The certificate file does not exist or you do not have permission to read that file.
- The certificate file that contains the certificate chain is not in PEM (base64 encoded) format.
- If you loaded a private key file before issuing this function, the private key in that file does not match the corresponding public key in the certificate.

#### Programming Considerations

- The first certificate in the file must be the certificate for your application. The next certificate in the file must be the certificate for the certificate authority (CA) that signed the certificate for your application. Subsequent certificates, if any exist, are for the CAs in the signing sequence.
- The entire list of certificates is passed to the remote node during the SSL handshake.
- Each SSL structure that is created from this CTX structure (by issuing the `SSL_new` function) inherits the certificates of that CTX structure.
- If you need only one certificate rather than a chain of certificates, you can use either the `SSL_CTX_use_certificate_file` or `SSL_use_certificate_file` function.

#### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

#### Related Information

- “`SSL_CTX_new`” on page 285
- “`SSL_CTX_new_shared`” on page 287
- “`SSL_CTX_use_certificate_file`” on page 298

## SSL\_CTX\_use\_certificate\_chain\_file

- “SSL\_new” on page 317
- “SSL\_use\_certificate\_file” on page 330.

### SSL\_CTX\_use\_certificate\_file

The `SSL_CTX_use_certificate_file` function loads the certificate for use with Secure Sockets Layer (SSL) sessions using a specific context (CTX) structure.

#### Format

```
#include <openssl/ssl.h>
int SSL_CTX_use_certificate_file(SSL_CTX *ctx, const char *file, int type)
```

##### ctx

A pointer to a token returned on the `SSL_CTX_new` call or the `SSL_CTX_new_shared` call.

##### file

A pointer to the name of the file that contains the certificate. The maximum length is 255 characters.

##### type

The file type, which is one of the following:

- |                          |                                                           |
|--------------------------|-----------------------------------------------------------|
| <b>SSL_FILETYPE_ASN1</b> | The file is in abstract syntax notation 1 (ASN.1) format. |
| <b>SSL_FILETYPE_PEM</b>  | The file is in PEM (base64 encoded) format.               |

#### Normal Return

Return code 1 indicates that the function was successful.

#### Error Return

A return code equal to 0 indicates an error. The following are the most likely causes of errors:

- The certificate file does not exist or you do not have permission to read that file.
- The file type is not valid. The file type must be abstract syntax notation 1 (ASN.1) or PEM (base64 encoded).
- If you loaded a private key file before issuing this function, the private key in that file does not match the corresponding public key in the certificate.

#### Programming Considerations

- Each SSL structure that is created from this CTX structure using the `SSL_new` function inherits the certificate of the CTX structure. You can override the certificate used by an individual SSL session by issuing the `SSL_use_certificate_file` function.
- If you are assigning many SSL sessions to the same CTX structure and all the sessions will use the same certificate, issue the `SSL_CTX_use_certificate_file` function once to assign the certificate to the CTX structure rather than issuing the `SSL_use_certificate_file` once for each SSL session.
- If you need to pass a chain of certificates rather than just one certificate, you must issue the `SSL_CTX_use_certificate_chain_file` function.

#### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## Related Information

- “SSL\_CTX\_new” on page 285
- “SSL\_CTX\_new\_shared” on page 287
- “SSL\_CTX\_use\_certificate\_chain\_file” on page 296
- “SSL\_CTX\_use\_certificate\_file” on page 298
- “SSL\_new” on page 317
- “SSL\_use\_certificate\_file” on page 330.

### SSL\_CTX\_use\_PrivateKey\_file

The `SSL_CTX_use_PrivateKey_file` function loads the private key for use with Secure Sockets Layer (SSL) sessions using a specific context (CTX) structure.

#### Format

```
#include <openssl/ssl.h>
```

```
int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx, const char *file, int type)
```

##### **ctx**

A pointer to a token returned on the `SSL_CTX_new` call or the `SSL_CTX_new_shared` call.

##### **file**

A pointer to the name of the file that contains the private key.

##### **type**

The file type, which must be the following:

**SSL\_FILETYPE\_PEM**                    The file is in PEM (base64 encoded) format.

#### Normal Return

Return code 1 indicates that the function was successful.

#### Error Return

A return code equal to 0 indicates an error. The following are the most likely causes of errors:

- The private key file does not exist or you do not have permission to read that file.
- The private key file is not in PEM (base64 encoded) format.
- If the private key file is encrypted, the password is not correct or no password was provided.
- If you loaded a certificate file before issuing this function, the public key in that certificate does not match the corresponding private key in the private key file.

#### Programming Considerations

- Before calling the `SSL_CTX_use_PrivateKey_file` function, you must identify the password for the private key file by issuing the `SSL_CTX_set_default_passwd_cb_userdata` function. Do this only if the private key file has been encrypted.
- If you are assigning many SSL sessions to the same CTX structure and all the sessions will use the same private key file, issue the `SSL_CTX_use_PrivateKey_file` function once to assign the certificate to the CTX structure rather than issuing the `SSL_CTX_use_PrivateKey_file` function once for each SSL session.

#### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

#### Related Information

- "SSL\_CTX\_new" on page 285
- "SSL\_CTX\_new\_shared" on page 287

## SSL\_CTX\_use\_PrivateKey\_file

- “SSL\_CTX\_set\_default\_passwd\_cb\_userdata” on page 293
- “SSL\_CTX\_use\_certificate\_chain\_file” on page 296
- “SSL\_use\_certificate\_file” on page 330.

### SSL\_CTX\_use\_RSAPrivateKey\_file

The `SSL_CTX_use_RSAPrivateKey_file` function loads the Rivest-Shamir-Adelman (RSA) private key for use with a Secure Sockets Layer (SSL) session using a specific context (CTX) structure.

#### Format

```
#include <openssl/ssl.h>
int SSL_CTX_use_RSAPrivateKey_file(SSL_CTX *ctx, const char *file, int type)
```

##### ctx

A pointer to a token returned on the `SSL_CTX_new` call or the `SSL_CTX_new_shared` call.

##### file

A pointer to the name of the file that contains the RSA private key. The maximum length is 255 characters.

##### type

The file type, which is one of the following:

<b>SSL_FILETYPE_ASN1</b>	The file is in abstract syntax notation 1 (ASN.1) format.
<b>SSL_FILETYPE_PEM</b>	The file is in PEM (base64 encoded) format.

#### Normal Return

Return code 1 indicates that the function was successful.

#### Error Return

A return code equal to 0 indicates an error. The following are the most likely causes of errors:

- The private key file does not exist or you do not have permission to read that file.
- The private key file is not in PEM (base64 encoded) format.
- If the private key file is encrypted, the password is not correct or no password was provided.
- If you loaded a certificate file before issuing this function, the public key in that certificate does not match the corresponding private key in the private key file.

#### Programming Considerations

- Before calling the `SSL_CTX_use_RSAPrivateKey_file` function, you must identify the password for the private key file, if the file is in PEM (base64 coded) format, by issuing the `SSL_CTX_set_default_passwd_cb_userdata` function. Do this only if the private key file has been encrypted.
- If you are assigning many SSL sessions to the same CTX structure and all the sessions will use the same private key file, issue the `SSL_CTX_use_RSAPrivateKey_file` function once to assign the certificate to the CTX structure rather than issuing the `SSL_CTX_use_RSAPrivateKey_file` function once for each SSL session.

#### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## **Related Information**

- “SSL\_CTX\_new” on page 285
- “SSL\_CTX\_new\_shared” on page 287
- “SSL\_CTX\_set\_default\_passwd\_cb\_userdata” on page 293
- “SSL\_CTX\_use\_certificate\_chain\_file” on page 296
- “SSL\_use\_certificate\_file” on page 330.

## SSL\_free

---

### SSL\_free

The SSL\_free function returns to the TPF system the Secure Sockets Layer (SSL) structure associated with an SSL session.

### Format

```
#include <openssl/ssl.h>
void SSL_free(SSL *ssl)
```

#### ssl

A pointer to a token returned on the SSL\_new call.

### Normal Return

None.

### Error Return

None.

### Programming Considerations

None.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

"SSL\_new" on page 317.

---

## SSL\_get\_cipher

The `SSL_get_cipher` function returns the name of the cipher associated with a specific Secure Sockets Layer (SSL) session.

### Format

```
#include <openssl/ssl.h>
const char *SSL_get_cipher(SSL *ssl)
```

**ssl**

A pointer to a token returned on the `SSL_new` call.

### Normal Return

Returns a pointer to the cipher name. Possible values are:

**NULL-MD5**

No data encryption; MD5 for message integrity.

**NULL-SHA**

No data encryption; SHA for message integrity.

**EXP-RC4-MD5**

Export RC4 (40-bit key) for data encryption; MD5 for message integrity.

**RC4-MD5**

RC4 (128-bit key) for data encryption; MD5 for message integrity.

**RC4-SHA**

RC4 (128-bit key) for data encryption; SHA for message integrity.

**EXP-RC2-CBC-MD5**

Export RC2 (40-bit key) for data encryption; MD5 for message integrity.

**EXP-RC4-MD5**

Export RC4 (40-bit key) for data encryption; MD5 for message integrity.

**EXP-DES-CBC-SHA**

Export DES (40-bit key) for data encryption; SHA for message integrity.

**DES-CBC3-SHA**

Triple-DES (168-bit key) for data encryption; SHA for message integrity.

**DES-CBC-MD5**

DES (56-bit key) for data encryption; MD5 for message integrity.

**DES-CBC3-MD5**

Triple-DES (168-bit key) for data encryption; MD5 for message integrity.

**RC2-CBC-MD5**

RC2 (128-bit key) for data encryption; MD5 for message integrity.

### Error Return

Returns a pointer to the text string **unknown**.

### Programming Considerations

The SSL session must be started before this function is issued, which means that the `SSL_connect` and `SSL_accept` functions must be issued before this function is issued.

## SSL\_get\_cipher

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “SSL\_accept” on page 274
- “SSL\_connect” on page 279
- “SSL\_new” on page 317.

---

## SSL\_get\_error

The `SSL_get_error` function returns information about why the previous Secure Sockets Layer (SSL) application programming interface (API) call resulted in an error return code.

### Format

```
#include <openssl/ssl.h>
int SSL_get_error(SSL *ssl,int ret)
```

**ssl**

A pointer to a token returned on the `SSL_new` call.

**ret**

The return code from the previous SSL API call.

### Normal Return

Returns one of the following values:

**SSL\_ERROR\_NONE**

No error to report. This is set when the value of the `ret` parameter is greater than 0.

**SSL\_ERROR\_SSL**

An error occurred in the SSL library.

**SSL\_ERROR\_WANT\_READ**

Processing was not completed successfully because there was no data available for reading, and the socket available for the SSL session is in nonblocking mode. Try the function again at a later time.

**SSL\_ERROR\_WANT\_WRITE**

Processing was not completed successfully because the socket associated with the SSL session is blocked from sending data. Try the function again at a later time.

**SSL\_ERROR\_SYSCALL**

An I/O error occurred. Issue the `sock_errno` function to determine the cause of the error.

**SSL\_ERROR\_ZERO\_RETURN**

The remote application shut down the SSL connection normally. Issue the `SSL_shutdown` function to shut down data flow for an SSL session.

**SSL\_ERROR\_WANT\_CONNECT**

Processing was not completed successfully because the SSL session was in the process of starting the session, but it has not completed yet. Try the function again at a later time.

### Error Return

None.

### Programming Considerations

- If an SSL API call results in an error return code, issue the `SSL_get_error` function for the following functions to obtain the reason for the error:
  - `SSL_accept`
  - `SSL_connect`
  - `SSL_read`

## SSL\_get\_error

- SSL\_shutdown
- SSL\_write.
- Do **not** use errno or the sock\_errno function to determine the cause of an SSL API error. Instead, you must use the SSL\_get\_error function. However, if you received the SSL\_ERROR\_SYSCALL return code after issuing the SSL\_get\_error function, it is appropriate to use the sock\_errno function.

## Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## Related Information

- “SSL\_accept” on page 274
- “SSL\_connect” on page 279
- “SSL\_read” on page 319
- “SSL\_shutdown” on page 329
- “SSL\_write” on page 334
- See *TPF Transmission Control Protocol/Internet Protocol* for more information about the errno and sock\_errno functions.

---

## SSL\_get\_peer\_certificate

The `SSL_get_peer_certificate` function returns the peer certificate that was received when the Secure Sockets Layer (SSL) session was started.

### Format

```
#include <openssl/ssl.h>
X509 *SSL_get_peer_certificate(SSL *ssl)
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call.

### Normal Return

Returns one of the following:

- A pointer to the certificate.
- NULL if no certificate was received when the SSL session started or the certificate is no longer available.
- NULL if issued with a shared SSL session as input.

### Error Return

None.

### Programming Considerations

The `SSL_get_peer_certificate` function checks to determine whether a peer certificate exists. Issue the `SSL_get_verify_result` function to determine whether the certificate is valid.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_get_verify_result`” on page 311
- “`SSL_new`” on page 317.

## SSL\_get\_session

---

### SSL\_get\_session

The `SSL_get_session` function returns a copy of the Secure Sockets Layer (SSL) session information for a specific SSL structure.

### Format

```
#include <openssl/ssl.h>
SSL_SESSION *SSL_get_session(SSL *ssl);
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call.

### Normal Return

Returns a pointer to an `SSL_SESSION` structure that contains information about the SSL session.

### Error Return

Returns `NULL` when no SSL session exists.

### Programming Considerations

- This function is useful only for client applications that want to use an SSL session again.
- Use the output of this function as input to the `SSL_set_session` function.
- You cannot use this function for shared SSL sessions.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_new`” on page 317
- “`SSL_set_session`” on page 326.

## SSL\_get\_verify\_result

The `SSL_get_verify_result` function returns the result of the remote peer certificate validation.

### Format

```
#include <openssl/ssl.h>
long SSL_get_verify_result(SSL *ssl)
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call

### Normal Return

Returns one of the following values:

#### **X509\_V\_OK**

The certificate was valid or no certificate was provided. Use the `SSL_get_peer_certificate` function to determine whether the certificate was provided or not.

#### **X509\_V\_ERR\_UNABLE\_TO\_GET\_ISSUER\_CERT**

Unable to find the certificate for one of the certificate authorities (CAs) in the signing hierarchy and that CA is not trusted by the local application.

#### **X509\_V\_ERR\_UNABLE\_TO\_DECRYPT\_CERT\_SIGNATURE**

Unable to decrypt the signature of the certificate.

#### **X509\_V\_ERR\_UNABLE\_TO\_DECODE\_ISSUER\_PUBLIC\_KEY**

The public key in the certificate could not be read.

#### **X509\_V\_ERR\_CERT\_SIGNATURE\_FAILURE**

The signature of the certificate is not valid.

#### **X509\_V\_ERR\_CERT\_NOT\_YET\_VALID**

The certificate is not valid until a date in the future.

#### **X509\_V\_ERR\_CERT\_HAS\_EXPIRED**

The certificate has expired.

#### **X509\_V\_ERR\_ERROR\_IN\_CERT\_NOT\_BEFORE\_FIELD**

There is a format error in the `notBefore` field of the certificate.

#### **X509\_V\_ERR\_ERROR\_IN\_CERT\_NOT\_AFTER\_FIELD**

There is a format error in the `notAfter` field of the certificate.

#### **X509\_V\_ERR\_DEPTH\_ZERO\_SELF\_SIGNED\_CERT**

The passed certificate is self-signed and the same certificate cannot be found in the list of trusted certificates.

#### **X509\_V\_ERR\_SELF\_SIGNED\_CERT\_IN\_CHAIN**

A self-signed certificate exists in the certificate chain. The certificate chain could be built up using the untrusted certificates, but the root CA could not be found locally.

#### **X509\_V\_ERR\_UNABLE\_TO\_GET\_ISSUER\_CERT\_LOCALLY**

The issuer certificate of a locally looked up certificate could not be found. This normally means that the list of trusted certificates is not complete.

#### **X509\_V\_ERR\_UNABLE\_TO\_VERIFY\_LEAF\_SIGNATURE**

No signatures could be verified because the certificate chain contains only one certificate, it is not self-signed, and the issuer is not trusted.

## SSL\_get\_verify\_result

### **X509\_V\_ERR\_INVALID\_CA**

A CA certificate is not valid because it is not a CA or its extensions are not consistent with the intended purpose.

### **X509\_V\_ERR\_PATH\_LENGTH\_EXCEEDED**

The **basicConstraints pathlength** parameter was exceeded.

### **X509\_V\_ERR\_INVALID\_PURPOSE**

The certificate that was provided cannot be used for its intended purpose.

### **X509\_V\_ERR\_CERT\_UNTRUSTED**

The root CA is not marked as trusted for its intended purpose.

### **X509\_V\_ERR\_CERT\_REJECTED**

The root CA is marked to reject the purpose specified.

### **X509\_V\_ERR\_SUBJECT\_ISSUER\_MISMATCH**

The issuer certificate was rejected because its subject name did not match the issuer name of the current certificate.

### **X509\_V\_ERR\_AKID\_SKID\_MISMATCH**

The issuer certificate was rejected because its subject key identifier was present and did not match the authority key identifier of the current certificate.

### **X509\_V\_ERR\_AKID\_ISSUER\_SERIAL\_MISMATCH**

The issuer certificate was rejected because its issuer name and serial number was present and did not match the authority key identifier of the current certificate.

### **X509\_V\_ERR\_KEYUSAGE\_NO\_CERTSIGN**

The issuer certificate was rejected because its keyUsage extension does not permit certificate signing.

### **X509\_V\_ERR\_CERT\_REVOKED**

The certificate was revoked by the issuer.

## Error Return

None.

## Programming Considerations

Client applications that have a verify mode of `SSL_VERIFY_NONE` must use the `SSL_get_verify_result` function to determine whether the certificate for the server application is valid or not.

## Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## Related Information

- “`SSL_get_peer_certificate`” on page 309
- “`SSL_new`” on page 317.

---

## SSL\_get\_version

The `SSL_get_version` function returns the protocol version of the current Secure Sockets Layer (SSL) connection.

### Format

```
#include <openssl/ssl.h>
const char * SSL_get_version(SSL *ssl)
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call

### Normal Return

Returns a character pointer to the name of the protocol version in use. Possible values are:

#### **SSLv2**

SSL version 2

#### **SSLv3**

SSL version 3

**TLSv1** TLS version 1

### Error Return

If the SSL session has not been started, it returns a pointer to the character string (**NONE**).

### Programming Considerations

Issue this call only after the SSL handshake has been completed.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

"`SSL_new`" on page 317.

## SSL\_library\_init

---

### SSL\_library\_init

The SSL\_library\_init function registers the available ciphers and message digests.

### Format

```
#include <openssl/ssl.h>
int SSL_library_init(void)
```

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

None.

### Programming Considerations

This is the first SSL function that an application issues before issuing any other SSL function.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

None.

---

## SSL\_load\_and\_set\_client\_CA\_list

The `SSL_load_and_set_client_CA_list` function loads certificates from a specific file and places the issuer name of each certificate in a specific Secure Sockets Layer (SSL) structure. Each certificate is for a certificate authority (CA) that the server application trusts and is willing to accept as the CA that issued the certificate of the remote client.

### Format

```
#include <openssl/ssl.h>
int SSL_load_and_set_client_CA_list(SSL *ssl, const char *file)
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call.

#### **file**

A pointer to the file that contains the certificates. The file must be in PEM (base64 encoded) format. The maximum length is 255 characters.

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

If unsuccessful, the `SSL_load_and_set_client_CA_list` function returns NULL. The following are the most likely causes of errors:

- The certificate authority (CA) file does not exist or you do not have permission to read that file.
- The CA file that contains the certificate chain is not in PEM (base64 encoded) format.

### Programming Considerations

- This function is unique to the TPF system.
- This function combines the `SSL_load_client_CA_file` and `SSL_set_client_CA_list` functions into one function. For shared SSL sessions, you must use this function rather than the `SSL_load_client_CA_file` and `SSL_set_client_CA_list` functions.
- This function is needed only by server applications that verify the identity of remote client applications when SSL sessions are started.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_load_client_CA_file`” on page 316
- “`SSL_new`” on page 317
- “`SSL_set_client_CA_list`” on page 324.

### SSL\_load\_client\_CA\_file

The `SSL_load_client_CA_file` function loads certificates from a specific file and returns the issuer name of each certificate.

### Format

```
#include <openssl/ssl.h>
STACK_OF(X509_NAME) *SSL_load_client_CA_file(const char *file)
```

#### file

A pointer to the name of the file that contains the certificates. The file must be in PEM (base64 encoded) format.

### Normal Return

Returns a stack of certificate issuer names. The names include each certificate authority (CA) that signed any of the certificates in the file.

### Error Return

If unsuccessful, the `SSL_load_client_CA_file` function returns NULL. The following are the most likely causes of errors:

- The certificate authority (CA) file does not exist or you do not have permission to read that file.
- The CA file that contains the certificate chain is not in PEM (base64 encoded) format.

### Programming Considerations

- The `SSL_load_client_CA_file` function is needed only by server applications that verify the identity of remote client applications when Secure Sockets Layer (SSL) sessions are started.
- The file must contain certificates for all CAs that the server application will accept as the CA that signed the certificate for the client application. This list of CAs is not necessarily the same list of CAs that the server application trusts.
- Pass the output of the `SSL_load_client_CA_file` function to the `SSL_CTX_set_client_CA_list` or the `SSL_set_client_CA_list` functions. The list of CAs will be sent to the client application when requesting its certificate.
- This function does not have an SSL or CTX structure as input. In addition, the `SSL_set_client_CA_list` function cannot be issued with a shared SSL session as input; therefore, you must use the `SSL_load_and_set_client_CA_list` function for shared SSL sessions.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_CTX_set_client_CA_list`” on page 292
- “`SSL_load_and_set_client_CA_list`” on page 315
- “`SSL_set_client_CA_list`” on page 324.

---

## SSL\_new

The `SSL_new` function creates a new Secure Sockets Layer (SSL) structure for use with an SSL session.

### Format

```
#include <openssl/ssl.h>
SSL *SSL_new(SSL_CTX *ctx)
```

#### **ctx**

A pointer to a token returned on the `SSL_CTX_new` call or the `SSL_CTX_new_shared` call.

### Normal Return

Returns a token that represents a new SSL structure.

### Error Return

A NULL pointer indicates an error.

## Programming Considerations

- Before calling the `SSL_new` function, you must call one of the following functions to set up the connection method:
  - `SSLv2_client_method`
  - `SSLv2_server_method`
  - `SSLv3_client_method`
  - `SSLv3_server_method`
  - `SSLv23_client_method`
  - `SSLv23_server_method`
  - `TLSv1_client_method`
  - `TLSv1_server_method`.
- Use the output of this function as input to subsequent functions that require an SSL structure as input.

## Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## Related Information

- “`SSL_CTX_new`” on page 285
- “`SSL_CTX_new_shared`” on page 287
- “`SSLv2_client_method`” on page 335
- “`SSLv2_server_method`” on page 336
- “`SSLv23_client_method`” on page 337
- “`SSLv23_server_method`” on page 338
- “`SSLv3_client_method`” on page 339
- “`SSLv3_server_method`” on page 340
- “`TLSv1_client_method`” on page 341
- “`TLSv1_server_method`” on page 342.

### SSL\_pending

The `SSL_pending` function returns the amount of data in the current Secure Sockets Layer (SSL) data record that is immediately available for reading on an SSL session.

### Format

```
#include <openssl/ssl.h>
int SSL_pending(SSL *ssl)
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call.

### Normal Return

Returns the number of bytes that are pending, which can be zero.

### Error Return

None.

### Programming Considerations

- Application data for an SSL session flows in SSL data records. An entire data record must be received from the network and processed by the SSL code before any data in that data record can be passed to the application when the `SSL_read` function is issued. If the amount of data in the data record is greater than the buffer size specified on the `SSL_read` function, only part of the data in the data record is passed to the application. The remaining data in that data record is pending and the amount of data that remains in the data record is the return code for the `SSL_pending` function.
- If the `SSL_pending` function returns a return code of 0, it does not necessarily mean that there is no data immediately available for reading on the SSL session. A return code of 0 indicates that there is no more data in the current SSL data record. However, more SSL data records may have been received from the network already. If the `SSL_pending` function returns a return code of 0, issue the `select` function, passing the file descriptor of the socket to check if the socket is readable. *Readable* means more data has been received from the network on the socket.

If the socket is readable, it does not necessarily mean that application data is available. The data on the socket could be SSL control information (such as an alert) rather than application data. The `select` function will also indicate that the socket is readable even if a partial SSL data record was received from the network.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_new`” on page 317
- “`SSL_read`” on page 319
- See *TPF Transmission Control Protocol/Internet Protocol* for more information about the `select` function.

---

## SSL\_read

The `SSL_read` function reads application data from a Secure Sockets Layer (SSL) session.

### Format

```
#include <openssl/ssl.h>
int SSL_read(SSL *ssl, char *buf, int num)
```

**ssl**

A pointer to a token returned on the `SSL_new` call.

**buf**

A pointer to the buffer into which to read the data.

**num**

The maximum number of bytes of data that the application can read.

### Normal Return

Returns the number of bytes of data (from 1 to the value specified on the **num** parameter) that are read.

### Error Return

A return code equal to 0 or a negative number indicates an error. Issue the `SSL_get_error` function to obtain specific information about the error.

### Programming Considerations

If this is a shared SSL session, the socket will be changed to nonblocking mode when the `SSL_read` function is completed.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_get_error`” on page 307
- “`SSL_new`” on page 317
- “`SSL_pending`” on page 318.

## SSL\_renegotiate

---

### SSL\_renegotiate

The `SSL_renegotiate` function creates a new set of cipher keys for an existing Secure Sockets Layer (SSL) session.

### Format

```
#include <openssl/ssl.h>
int SSL_renegotiate(SSL *ssl)
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

A return code equal to 0 indicates an error.

### Programming Considerations

This function is useful for long-running SSL sessions to create a new set of cipher keys periodically.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

"`SSL_new`" on page 317.

## SSL\_set\_cipher\_list

The `SSL_set_cipher_list` function sets the ciphers for use by a specific Secure Sockets Layer (SSL) session that session that is started using the specified SSL structure.

### Format

```
#include <openssl/ssl.h>
int ssl_set_cipher_list(SSL *ssl, const char *str)
```

#### ssl

A pointer to a token returned on the `SSL_new` call.

#### str

A pointer to a string that contains one or more ciphers separated by a colon, comma, or blank. The maximum length is 255 characters.

You must specify the ciphers in order of preference from highest to lowest. The TPF system supports the following SSL version 3 and Transport Layer Security (TLS) version 1 ciphers that are used by the Rivest-Shamir-Adelman (RSA) key exchange:

<b>NULL-MD5</b>	No data encryption; MD5 for message integrity.
<b>NULL-SHA</b>	No data encryption; SHA for message integrity.
<b>EXP-RC4-MD5</b>	Export RC4 (40-bit key) for data encryption; MD5 for message integrity.
<b>RC4-MD5</b>	RC4 (128-bit key) for data encryption; MD5 for message integrity.
<b>RC4-SHA</b>	RC4 (128-bit key) for data encryption; SHA for message integrity.
<b>EXP-RC2-CBC-MD5</b>	Export RC2 (40-bit key) for data encryption; MD5 for message integrity.
<b>EXP-DES-CBC-SHA</b>	Export DES (40-bit key) for data encryption; SHA for message integrity.
<b>DES-CBC-SHA</b>	DES (56-bit key) for data encryption; SHA for message integrity.
<b>DES-CBC3-SHA</b>	Triple-DES (168-bit key) for data encryption; SHA for message integrity.

The TPF system supports the following SSL version 2 ciphers that are used by the RSA key exchange:

<b>RC4-MD5</b>	RC4 (128-bit key) for data encryption; MD5 for message integrity.
<b>EXP-RC4-MD5</b>	Export RC4 (40-bit key) for data encryption; MD5 for message integrity.
<b>RC2-CBC-MD5</b>	RC2 (128-bit key) for data encryption; MD5 for message integrity.
<b>EXP-RC2-CBC-MD5</b>	Export RC2 (40-bit key) for data encryption; MD5 for message integrity.
<b>DES-CBC-MD5</b>	DES (56-bit key) for data encryption; MD5 for message integrity.

## SSL\_set\_cipher\_list

### DES-CBC3-MD5

Triple-DES (168-bit key) for data encryption;  
MD5 for message integrity.

## Normal Return

Return code 1 indicates that the function was successful.

## Error Return

A return code equal to 0 indicates an error.

## Programming Considerations

- When an SSL structure is first created using the `SSL_new` function, the structure inherits the cipher list assigned to the context (CTX) structure that was used to create the SSL structure. The `SSL_set_cipher_list` function overrides that cipher list for a specific SSL structure.
- If you are assigning many SSL sessions to the same CTX structure and each session will use the same cipher list, issue the `SSL_CTX_set_cipher_list` function once to assign the cipher list to the CTX structure rather than issuing the `SSL_set_cipher_list` function once for each SSL session.
- If you start an SSL session without issuing the `SSL_CTX_set_cipher_list` or the `SSL_set_cipher_list` function, the system default cipher list is used.

The default ciphers for SSL version 2 are:

- DES-CBC-MD5
- DES-CBC3-MD5
- EXP-RC2-CBC-MD5
- EXP-RC4-MD5
- RC2-CBC-MD5
- RC4-MD5.

The default ciphers for SSL version 3 are:

- DES-CBC-SHA
- DES-CBC3-SHA
- EXP-DES-CBC-SHA
- EXP-RC2-CBC-MD5
- EXP-RC4-MD5
- RC4-SHA.

The default ciphers for TLS version 1 are:

- DES-CBC3-SHA
- DES-CBC-SHA
- EXP-DES-CBC-SHA
- EXP-RC2-CBC-MD5
- EXP-RC4-MD5
- RC4-MD5
- RC4-SHA.

## Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## Related Information

- "SSL\_CTX\_set\_cipher\_list" on page 289
- "SSL\_new" on page 317.

### SSL\_set\_client\_CA\_list

The `SSL_set_client_CA_list` function identifies the list of certificate authorities (CAs) that are sent to the remote client application when requesting the client certificates for a specific Secure Sockets Layer (SSL) session. The client application must provide a certificate that was signed by one of the CAs in the list.

### Format

```
#include <openssl/ssl.h>
void SSL_set_client_CA_list(SSL *ssl,STACK_OF(X509_NAME) *list)
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call.

#### **list**

A pointer to a stack of CA names.

### Normal Return

None.

### Error Return

None.

### Programming Considerations

- The `SSL_set_client_CA_list` function is needed only by server applications that verify the identity of remote client applications when SSL sessions are started.
- Use the output from the `SSL_load_client_CA_file` function as input to this function.
- If the `SSL_set_client_CA_list` function is not used and you request a client certificate, the list of CA names that get passed to the client application are the CAs from the `SSL_CTX_load_verify_locations` function.
- This function cannot be issued with a shared SSL session as input. In addition, the `SSL_load_client_CA_file` function does not have an SSL or CTX structure as input; therefore, you must use the `SSL_load_and_set_client_CA_list` function for shared SSL sessions.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_CTX_load_verify_locations`” on page 283
- “`SSL_load_client_CA_file`” on page 316
- “`SSL_load_and_set_client_CA_list`” on page 315
- “`SSL_new`” on page 317.

---

## SSL\_set\_fd

The `SSL_set_fd` function assigns a socket to a Secure Sockets Layer (SSL) structure.

### Format

```
#include <openssl/ssl.h>
int SSL_set_fd(SSL *ssl, int fd)
```

**ssl**

A pointer to a token returned on the `SSL_new` call.

**fd** The file descriptor of the socket.

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

A return code equal to 0 indicates an error.

### Programming Considerations

Assign the socket to an SSL structure before starting the SSL session with the `SSL_connect` or `SSL_accept` function. You can assign the socket to an SSL structure anytime after the Transmission Control Protocol (TCP) connection is established, meaning after the connect or accept functions is completed successfully.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_accept`” on page 274
- “`SSL_connect`” on page 279
- “`SSL_new`” on page 317
- See the *Socket Application Programming Interface Overview* section of *TPF Transmission Control Protocol/Internet Protocol* for more information about the connect and accept functions.

### SSL\_set\_session

The `SSL_set_session` function sets up Secure Sockets Layer (SSL) session information for use by the `SSL_connect` function when reusing an SSL session.

### Format

```
#include <openssl/ssl.h>
int SSL_set_session(SSL *ssl, SSL_SESSION *session);
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call.

#### **session**

A pointer to the `SSL_SESSION` structure.

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

A return code equal to 0 indicates an error.

### Programming Considerations

- This function is useful only for client applications that want to reuse an SSL session.
- The output of the `SSL_get_session` function is an `SSL_SESSION` structure. Use the output of the `SSL_get_session` function as input to the `SSL_set_session` function.
- You cannot use this function for shared SSL sessions.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_connect`” on page 279
- “`SSL_get_session`” on page 310
- “`SSL_new`” on page 317.

## SSL\_set\_verify

The `SSL_set_verify` function indicates whether to verify the identity of the remote application or not when the Secure Sockets Layer (SSL) session is started.

### Format

```
#include <openssl/ssl.h>
void SSL_set_verify(SSL *ssl, int mode, int (*cb)
                  (int ok, X509_STORE_CTX *ctx))
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call.

#### **mode**

One or more of the following verify options:

##### **SSL\_VERIFY\_NONE**

Use this option if you do not want to verify the identity of the remote peer. This option must be used alone; no other options can be specified. Consider the following when using this option:

- If the application is a server, the application will not request the certificate for the remote client application when the SSL session is started.
- If the application is a client, the certificate for the remote server application will be validated; however, the SSL session will be started regardless of whether or not the certificate for the remote server application is valid. Issue the `SSL_get_verify_result` function to check whether or not the certificate for the server is valid.

##### **SSL\_VERIFY\_PEER**

Use this option to verify the identify of the remote peer when the SSL session is started. Consider the following when using this option:

- If the application is a server, the application will request and verify the certificate for the remote client application when the SSL session is started. If the remote client application provides a certificate that is not valid, the SSL session fails.
- If the application is a client, the certificate for the remote server application is validated. If the certificate for the remote server application is not valid, the SSL session fails.

##### **SSL\_VERIFY\_FAIL\_IF\_NO\_PEER\_CERT**

Use this option to request that the remote client application send its certificate when the SSL session is starting, and to end the SSL session if no certificate is provided. This option only has meaning if the `SSL_VERIFY_PEER` option is also set. Do not use this option when your application is the client.

##### **SSL\_VERIFY\_CLIENT\_ONCE**

Use this option to verify the identity of the remote client application when the SSL session is first started. If the SSL session is renegotiated, do not verify the identify of the client application again. This option only has meaning if the `SSL_VERIFY_PEER` option is also set. Do not use this option when your application is the client.

**cb** A pointer set to `NULL`.

### Normal Return

None.

## SSL\_set\_verify

### Error Return

None.

### Programming Considerations

The default value for the verify mode is `SSL_VERIFY_NONE` when the `CTX` structure is created. You must issue the `SSL_set_verify` function or the `SSL_CTX_set_verify` function with the `SSL_VERIFY_PEER` option if you want to authenticate remote peers when SSL sessions are started.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_CTX_set_verify`” on page 294
- “`SSL_get_verify_result`” on page 311
- “`SSL_new`” on page 317.

---

## SSL\_shutdown

The `SSL_shutdown` function shuts down data flow for a Secure Sockets Layer (SSL) session.

### Format

```
#include <openssl/ssl.h>
int SSL_shutdown(SSL *ssl)
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call.

### Normal Return

- Return code 0 indicates that the application issued the `SSL_shutdown` function first. Continue issuing the `SSL_shutdown` function until you receive return code 1, which indicates the remote application has also shut down.
- In SSL version 3 and TLS version 1, return code 1 indicates that both the client and server applications have issued the `SSL_shutdown` function.
- In SSL version 2, a return code of 1 is always returned.

### Error Return

A return code equal to `-1` indicates an error. Issue the `SSL_get_error` function to obtain specific information about the error.

### Programming Considerations

- The `SSL_shutdown` function is the normal way to shut down an SSL session. It is a good idea that you shut down an SSL session before the socket is shut down and closed.
- An alert is sent to the remote partner to notify it that the connection is ending normally. Normal shutdown is required if you want to resume the SSL session across a different socket at a later time.
- Both the client and server applications must issue the `SSL_shutdown` function to shut down the connection normally.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_get_error`” on page 307
- “`SSL_new`” on page 317.

### SSL\_use\_certificate\_file

The `SSL_use_certificate_file` function loads the certificate for use with a Secure Sockets Layer (SSL) session.

### Format

```
#include <openssl/ssl.h>
int SSL_use_certificate_file(SSL *ssl, const char *file, int type)
```

#### **ssl**

A pointer to a token returned on the `SSL_new` call.

#### **file**

A pointer to the name of the file that contains the certificate. The maximum length is 255 characters.

#### **type**

The file type, which is one of the following:

**SSL\_FILETYPE\_ASN1**            The file is in abstract syntax notation 1 (ASN.1) format.

**SSL\_FILETYPE\_PEM**            The file is in PEM (base64 encoded) format.

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

- A return code equal to 0 indicates an error. The following are the most likely causes of errors:
  - The certificate file does not exist or you do not have permission to read that file.
  - The file type is not valid. The file type must be ASN.1 or PEM.
- If you loaded a private key file before issuing this function, the private key in that file does not match the corresponding public key in the certificate.

### Programming Considerations

- When an SSL structure is first created using the `SSL_new` function, the structure inherits the certificate (if any exists) that is assigned to the context (CTX) structure that was used to create the SSL structure. The `SSL_use_certificate_file` function allows you to use a certificate other than the one assigned to the CTX structure.
- If you are assigning many SSL sessions to the same CTX structure and all sessions will use the same certificate, issue the `SSL_CTX_use_certificate_file` function once to assign the certificate to the CTX structure rather than issuing the `SSL_use_certificate_file` once for each SSL session.
- If you need to pass a chain of certificates rather than just one certificate, you must issue the `SSL_CTX_use_certificate_chain_file` function.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

## Related Information

- “SSL\_CTX\_use\_certificate\_chain\_file” on page 296
- “SSL\_CTX\_use\_certificate\_file” on page 298
- “SSL\_new” on page 317.

### SSL\_use\_PrivateKey\_file

The `SSL_use_PrivateKey_file` function loads the private key for use with a Secure Sockets Layer (SSL) session.

#### Format

```
#include <openssl/ssl.h>
int SSL_use_PrivateKey_file(SSL *ssl, const char *file, int type)
```

##### **ssl**

A pointer to a token returned on the `SSL_new` call.

##### **file**

A pointer to the name of the file that contains the private key. The maximum length is 255 characters.

##### **type**

The type of file, which must be the following:

**SSL\_FILETYPE\_PEM**            The file is in PEM (base64 encoded) format.

#### Normal Return

Return code 1 indicates that the function was successful.

#### Error Return

A return code equal to 0 indicates an error. The following are the most likely causes of errors:

- The private key file does not exist or you do not have permission to read that file.
- The private key file is not in PEM (base64 encoded) format.
- If the private key file is encrypted, the password is not correct or no password was provided.
- If you loaded a certificate file before issuing this function, the public key in that certificate does not match the corresponding private key in the private key file.

#### Programming Considerations

Before calling the `SSL_use_PrivateKey_file` function, you must identify the password for the private key file by issuing the `SSL_CTX_set_default_passwd_cb_userdata` function. Do this only if the private key file has been encrypted.

#### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

#### Related Information

- “`SSL_CTX_set_default_passwd_cb_userdata`” on page 293
- “`SSL_new`” on page 317.

---

## SSL\_use\_RSAPrivateKey\_file

The `SSL_use_RSAPrivateKey_file` function loads the Rivest-Shamir-Adelman (RSA) private key for use with a Secure Sockets Layer (SSL) session.

### Format

```
#include <openssl/ssl.h>
int SSL_use_RSAPrivateKey_file(SSL *ssl, const char *file, int type)
```

**ssl**

A pointer to a token returned on the `SSL_new` call.

**file**

A pointer to the name of the file that contains the RSA private key. The maximum length is 255 characters.

**type**

The file type, which is one of the following:

<b>SSL_FILETYPE_ASN1</b>	The file is in abstract syntax notation 1 (ASN.1) format.
<b>SSL_FILETYPE_PEM</b>	The file is in PEM (base64 encoded) format.

### Normal Return

Return code 1 indicates that the function was successful.

### Error Return

A return code equal to 0 indicates an error. The following are the most likely causes of errors:

- The private key file does not exist or you do not have permission to read that file.
- The private key file is not in PEM (base64 encoded) format.
- If the private key file is encrypted, the password is not correct or no password was provided.
- If you loaded a certificate file before issuing this function, the public key in that certificate does not match the corresponding private key in the private key file.

### Programming Considerations

Before calling the `SSL_use_RSAPrivateKey_file` function, you must identify the password for the private key file, if the file is in PEM (base64 coded) format, by issuing the `SSL_CTX_set_default_passwd_cb_userdata` function. Do this only if the private key file has been encrypted.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_CTX_set_default_passwd_cb_userdata`” on page 293
- “`SSL_new`” on page 317.

### SSL\_write

The `SSL_write` function writes application data across a Secure Sockets Layer (SSL) session.

### Format

```
#include <openssl/ssl.h>
int SSL_write(SSL *ssl, const char *buf, int num)
```

**ssl**

A pointer to a token returned on the `SSL_new` call.

**buf**

A pointer to the data to send.

**num**

The number of bytes of data to send. The maximum number of byte that can be sent is 32 000 for sessions using SSL version 2.

### Normal Return

Returns the number of bytes of data (from 1 to the value specified on the **num** parameter) sent.

### Error Return

A return code equal to 0 or a negative number indicates an error. Issue the `SSL_get_error` function to obtain specific information about the error.

### Programming Considerations

- Normally, the `SSL_write` function processes all data passed by the application, in which case the return code is equal to the value specified for the **num** parameter. However, if the socket becomes blocked during processing, it is possible that only some of the data will be processed, in which case the return code is greater than 0, but less than the value specified for the **num** parameter. When this occurs, you must adjust the value specified on the **buf** parameter and the value specified for the **num** parameter, and then issue the `SSL_write` function again to send the remaining data.
- If this is a shared SSL session, the socket will be changed to nonblocking mode when the `SSL_write` function is completed.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

"`SSL_new`" on page 317.

---

## SSLv2\_client\_method

The `SSLv2_client_method` function is used to indicate that the application is a client and supports Secure Sockets Layer version 2 (SSLv2).

### Format

```
#include <openssl/ssl.h>
SSL_METHOD *SSLv2_client_method(void)
```

### Normal Return

A pointer to the appropriate connection method.

### Error Return

None.

### Programming Considerations

- When an SSL session is started, an SSLv2 CLIENT\_HELLO command is sent to indicate that the client application only supports SSL version 2. The remote server application must also support SSL version 2 for the SSL session to be established.
- Use the output of this function as input to the `SSL_CTX_new` function or the `SSL_CTX_new_shared` function.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- "SSL\_CTX\_new" on page 285
- "SSL\_CTX\_new\_shared" on page 287.

## SSLv2\_server\_method

---

### SSLv2\_server\_method

The `SSLv2_server_method` function indicates that the application is a server and supports Secure Sockets Layer version 2 (SSLv2).

### Format

```
#include <openssl/ssl.h>
SSL_METHOD *SSLv2_server_method(void)
```

### Normal Return

A pointer to the appropriate connection method.

### Error Return

None.

### Programming Considerations

- The server application only understands SSLv2 CLIENT\_HELLO commands; therefore, the remote client applications must support SSL version 2 and send SSLv2 CLIENT\_HELLO commands to the server application. The server application will send back a SERVER\_HELLO command that indicates that it only supports SSL version 2.
- Use the output of this function as input to the `SSL_CTX_new` function or the `SSL_CTX_new_shared` function.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- "SSL\_CTX\_new" on page 285
- "SSL\_CTX\_new\_shared" on page 287.

---

## SSLv23\_client\_method

The `SSLv23_client_method` function indicates that the application is a client and supports Secure Sockets Layer version 2 (SSLv2), Secure Sockets Layer version 3 (SSLv3), and Transport Layer Security version 1 (TLSv1).

### Format

```
#include <openssl/ssl.h>
SSL_METHOD *SSLv23_client_method(void)
```

### Normal Return

A pointer to the appropriate connection method.

### Error Return

None.

### Programming Considerations

- When an SSL session is started, an SSLv2 CLIENT\_HELLO command is sent and indicates that the client application also supports SSLv3 and TLSv1. This client application can establish connections with any remote server application that supports SSLv2 and, optionally, other versions.
- Issue the `SSLv23_client_method` function if you do not know which SSL versions the remote server application supports, or if the SSL versions supported by the server application are likely to change.
- Use the output of this function as input to the `SSL_CTX_new` function or the `SSL_CTX_new_shared` function.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_CTX_new`” on page 285
- “`SSL_CTX_new_shared`” on page 287.

### SSLv23\_server\_method

The `SSLv23_server_method` function indicates that the application is a server and supports Secure Sockets Layer version 2 (SSLv2), Secure Sockets Layer version 3 (SSLv3), and Transport Layer Security version 1 (TLSv1).

### Format

```
#include <openssl/ssl.h>
SSL_METHOD *SSLv23_server_method(void)
```

### Normal Return

A pointer to the appropriate connection method.

### Error Return

None.

### Programming Considerations

- The server application understands SSLv2, SSLv3, and TLSv1 `CLIENT_HELLO` commands; therefore, any remote client application that supports SSL can connect to the server application.
- Use the `SSLv23_server_method` function when remote client applications using different SSL versions will be connecting to the server application.
- Use the output of this function as input to the `SSL_CTX_new` function or the `SSL_CTX_new_shared` function.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- “`SSL_CTX_new`” on page 285
- “`SSL_CTX_new_shared`” on page 287.

---

## SSLv3\_client\_method

The `SSLv3_client_method` function indicates that the application is a client and supports Secure Sockets Layer version 3 (SSLv3).

### Format

```
#include <openssl/ssl.h>
SSL_METHOD *SSLv3_client_method(void)
```

### Normal Return

A pointer to the appropriate connection method.

### Error Return

None.

### Programming Considerations

- When an SSL session is started, an SSLv3 CLIENT\_HELLO command is sent and indicates that the client application only supports SSL version 3. The remote server application must also support SSLv3 for the SSL session to be established. If the remote server application only supports SSL version 2 (SSLv2), the SSL session will fail.
- Use the output of this function as input to the `SSL_CTX_new` function or the `SSL_CTX_new_shared` function.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- "SSL\_CTX\_new" on page 285
- "SSL\_CTX\_new\_shared" on page 287.

### SSLv3\_server\_method

The `SSLv3_server_method` function indicates that the application is a server and supports Secure Sockets Layer version 3 (SSLv3).

### Format

```
#include <openssl/ssl.h>
SSL_METHOD *SSLv3_server_method(void)
```

### Normal Return

A pointer to the appropriate connection method.

### Error Return

None.

### Programming Considerations

- The server application only understands SSLv3 CLIENT\_HELLO commands; therefore, remote client applications must support SSL version 3 and send SSLv3 CLIENT\_HELLO commands to the server application. The server application sends back a SERVER\_HELLO command that indicates that it only supports SSLv3. Remote client applications that only support SSL version 2 (SSLv2) cannot connect to this server application.
- Use the output of this function as input to the `SSL_CTX_new` function or the `SSL_CTX_new_shared` function.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- "SSL\_CTX\_new" on page 285
- "SSL\_CTX\_new\_shared" on page 287.

---

## TLV1\_client\_method

The TLV1\_client\_method function indicates that the application is a client and supports Transport Layer Security version 1 (TLV1).

### Format

```
#include <openssl/ssl.h>
SSL_METHOD *TLV1_client_method(void)
```

### Normal Return

A pointer to the appropriate connection method.

### Error Return

None.

### Programming Considerations

- When a Secure Sockets Layer (SSL) session is started, a TLV1 CLIENT\_HELLO command is sent to indicate that the client application only supports TLV1. The remote server application must also support TLV1 for the SSL session to be established.
- Use the output of this function as input to the SSL\_CTX\_new function or the SSL\_CTX\_new\_shared function.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- "SSL\_CTX\_new" on page 285
- "SSL\_CTX\_new\_shared" on page 287.

### TLV1\_server\_method

The TLV1\_server\_method function indicates that the application is a server and supports Transport Layer Security version 1 (TLV1).

### Format

```
#include <openssl/ssl.h>
SSL_METHOD *TLV1_server_method(void)
```

### Normal Return

A pointer to the appropriate connection method.

### Error Return

None.

### Programming Considerations

- The server application only understands TLV1 CLIENT\_HELLO commands; therefore, remote client applications must support TLV1 and send TLV1 CLIENT\_HELLO commands to the server application. The server application will not understand or accept SSLv2 and SSLv3 CLIENT\_HELLO commands.
- Use the output of this function as input to the SSL\_CTX\_new function or the SSL\_CTX\_new\_shared function.

### Examples

For sample SSL applications, go to <http://www.ibm.com/tpf/pubs/tpfpubs.htm>, click **SSL for the TPF 4.1 System: An Online User's Guide**, and click **Examples** from the left navigation bar.

### Related Information

- "SSL\_CTX\_new" on page 285
- "SSL\_CTX\_new\_shared" on page 287.

---

## Appendix A. CLAW Trace Postprocessor

This appendix provides the following information:

- Sample job control language (JCL) for the Common Link Access to Workstation (CLAW) data trace postprocessor
- Sample data trace output
- Sample JCL for the CLAW process trace postprocessor
- Sample process trace output.

---

### Sample JCL for the CLAW Data Trace Postprocessor

The CLAW data trace postprocessor (CLTD) program reads an RTA/RTL log tape created by the CLAW online trace function. The CLAW online trace function is activated by the ZCLAW TRACE command. The CLTD program is designed to run in an offline environment.

Use the CLTD program to generate a report of logged message entries to an output device such as a printer. Figure 32 contains sample JCL that you can use to run the CLAW data trace postprocessor.

```
//FORMAT JOB (GMICO,5991C),B117,MSGLEVEL=(1,1),CLASS=A
/*ROUTE PRINT SYSTEM(USERID)
/*ROUTE PUNCH SYSTEM(USERID)
//*****
//***                                     ***
//***                                     ***
//***                                     ***
//*****
//A          EXEC PGM=CLTD40
//STEPLIB DD DSN=ACPO000.DEVP.TEST.LK,DISP=SHR
//LOGTAPE DD DISP=OLD,UNIT=VTAPE,VOL=SER=A00147,
//          DSN=RTA.TAPE
//REPORT DD SYSOUT=A
//TERM#OUT DD SYSOUT=A
//TERM#IN DD *
PRINT BLK=1000,MSG=43,DATE=11FEB,TIME=19.27.58,DUR=00.06
//TERM#IN DD *
DEFAULT
```

Figure 32. Sample JCL for the CLAW Data Trace Postprocessor

Input parameters to the CLTD program must be in 80-column format starting with column 1. You can enter the following parameters as the JCL input data set:

#### PRINT

The PRINT parameter must be displayed in column 1 using a space to separate it from its keyword = argument options list. Each parameter must be separated by a comma. The PRINT parameter is required if you want to specify the following options:

#### BLK=nnnn

Block record criteria, where *nnnn* is a number from 0001 to 9999. Each block element must be right-justified with zero padding. The program processes only the first *nnnn* CLAW 4-K trace blocks. The default block record criteria is 1000.

**MSG=*nn***

Message entry criteria, where *nn* is a number from 01 to 63. Each message element must be right-justified with zero padding. As many as *nn* messages in each 4-K trace block is processed.

**DATE=*ddmmm***

Date criteria, where *dd* is day 01 to 31 of a month and *mmm* is the month in 3-character format (for example, JAN, FEB, MAR, APR, MAY, JUN JUL, AUG, SEP, OCT, NOV, DEC). The CLTD program processes only those CLAW trace blocks that match the date requested. If the date is not provided, all blocks are processed in the BLK and MSG parameters.

**TIME=*hh.mm.ss***

Start time criteria, where *hh.mm.ss* is hours, minutes, and seconds in continental time format (for example, 15.00.00). Each time element must be right-justified with zero padding. The CLTD program processes only those CLAW trace blocks traced at or after the requested time. The DATE parameter is a prerequisite. If the TIME parameter is not provided, all blocks in the specified date are processed in the BLK and MSG parameters.

**DUR=*mm.ss***

Duration criteria, where *mm* is minute 00 to 59 and *ss* is second 00 to 59. Each time element must be right-justified with zero padding. If requested, the DATE and TIME parameters are required. You can specify an ending time to be used once the program starts processing the CLAW trace blocks that meet the start time criteria. Processing ends once a CLAW trace block whose 4-K block time stamp exceeds the duration period. A duration of minutes that carries to the next day will be set to the maximum of 59 minutes of the 23rd hour of the requested day. If the DUR parameter is not specified and the TIME parameter is specified, all blocks whose date and time stamps meet the criteria are processed.

**DEFAULT**

Use the DEFAULT parameter if you do not want to specify the PRINT parameter with options.

---

## CLAW Data Trace Postprocessor

Figure 33 on page 346 provides a sample of data output that is created by the Common Link Access to Workstation (CLAW) data trace postprocessor.

The output contains 64-byte entries of trace data, each representing data that has been sent to, or received from, the TCP/IP offload device.

- The 64-byte entries whose first 4 bytes are equal to X'00010001' represent socket application programming interface (API) requests sent to the TCP/IP offload device from the TPF system.
- The 64-byte entries whose first 4 bytes are equal to X'00010002' represent socket API responses received from the TCP/IP offload device by the TPF system.
- The 64-byte entries whose first 4 bytes are equal to X'00010009' represent unsolicited messages received from the TCP/IP offload device by the TPF system.

The trace data for this particular report represents a stream socket session in which the TPF system is the server and an OS/2 workstation is the client. During this session, a TPF socket application has issued socket, bind, listen, accept, read,

send, write, shutdown, and close socket API functions to the TCP/IP offload device. The trace data shows that each socket API request sent to the TCP/IP offload device is associated with a socket API response from the TCP/IP offload device. During this session, the TCP/IP offload device has also sent several unsolicited messages to the TPF system to indicate a change in the state of the stream socket connection. The `c$isiucv.h` header file and the `ISIUCV` data macro provide information about how to interpret the individual 64-byte trace entries printed by the CLAW data trace postprocessor.

See “Using the CLAW Data Trace Function” on page 32 for more information about the CLAW data trace function.



CLAW Trace / Log Tape Report

ADDR:  
 DATE: 11FEB            ID: TD, x'E3C4'            LAST: 3848  
 TIME: 19.27.58        SWITCH: x'F0'                NEXT: 3996

```

-----
                XMIT TIME: 46.24        READ ADDR:0200
MSG=x'000100090B0072005A29EF0000000198000001001E00000000000010F3030000'
MSG=x'0000001E00000001000000000000198000000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.25        READ ADDR:0200
MSG=x'0001000101A52D000073C0000001019800000000000002100000000000018'
MSG=x'000000000000000000000000000000000000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.36        READ ADDR:0200
MSG=x'0001000201A52D000073C000000101980000000000000210000001800000000'
MSG=x'00000197019800000200041209756B56C0000021CDB508A03061626364656667'
      SIZE = 0148

                XMIT TIME: 46.36        READ ADDR:0200
MSG=x'0001000101A5300000000000000E01970000000000000000000000000008000'
MSG=x'000000000000000000000000000000000000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.36        READ ADDR:0200
MSG=x'0001000101A52D000073C0000001019800000000000002200000000000018'
MSG=x'000000000000000000000000000000000000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.37        READ ADDR:0200
MSG=x'0001000201A5300000000000000E0197000000000000000000000250B000000'
MSG=x'000000050000000000000000000000000000000000000000000000000000'
      SIZE = 0148
-----
  
```

Figure 33. Sample Output from the CLAW Data Trace Postprocessor (Part 2 of 5)

CLAW Trace / Log Tape Report

	ADDR:		
DATE: 11FEB	ID: TD, x'E3C4'	LAST: 3848	
TIME: 19.28.11	SWITCH: x'F0'	NEXT: 3996	

```

-----
                XMIT TIME: 46.37      READ ADDR:0200
MSG=x'0001000101A530000081D000001401970000000000000000000001800000008'
MSG=x'00000000000000000000000000000000000000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.37      READ ADDR:0200
MSG=x'0001000201A530000081D0000014019700000000000000000000080E000000'
MSG=x'00000004000000000000000000000000000000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.37      READ ADDR:0200
MSG=x'0001000101A530000081D000000E0197000000000000000000000000008000'
MSG=x'00000000000000000000000000000000000000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.37      READ ADDR:0200
MSG=x'0001000201A530000081D000000E01970000000000000000000005D40B000000'
MSG=x'000005B40000B7A4F9FF00000021CC7E4D7A0000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.37      READ ADDR:0200
MSG=x'0001000101A530000081D000000E0197000000000000000000000000008000'
MSG=x'00000000000000000000000000000000000000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.37      READ ADDR:0200
MSG=x'0001000201A530000081D000000E01970000000000000000000005D40B000000'
MSG=x'000005B4000201A530000081D000000E01970000000000000000000000000000'
      SIZE = 0148
-----

```

Figure 33. Sample Output from the CLAW Data Trace Postprocessor (Part 3 of 5)



## CLAW Trace / Log Tape Report

ADDR:  
 DATE: 11FEB ID: TD, x'E3C4' LAST: 3848  
 TIME: 19.28.11 SWITCH: x'F0' NEXT: 3996

```

-----
                XMIT TIME: 46.37      READ ADDR:0200
MSG=x'000100090B0072005A29EF000000FFFF000001002100000000000010F4030000'
MSG=x'000000210000000400000000FFFF000000000000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.37      READ ADDR:0200
MSG=x'0001000101A530000081D00000030197000000000000000000000000000000008'
MSG=x'00000000000000000000000000000000000000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.37      READ ADDR:0200
MSG=x'000100090B0072005A29EF000000FFFF000001002100000000000010F5030000'
MSG=x'000000210000000000000000FFFF000000000000000000000000000000000000'
      SIZE = 0148

                XMIT TIME: 46.38      READ ADDR:0200
MSG=x'0001000201A530000081D0000003019700000000000000000000000802000000'
MSG=x'00000000B7008001000000000000019800000000000000000000000000000000'
      SIZE = 0148
-----

```

Figure 33. Sample Output from the CLAW Data Trace Postprocessor (Part 5 of 5)

## Sample JCL for the CLAW Process Trace Postprocessor

Use the CLAW process trace postprocessor (CLTP) to format and print the CLAW process trace information. Figure 34 contains sample JCL that you can use to run the CLAW process trace postprocessor.

```

//FORMAT JOB (GMICO,5991C),B117,MSGLEVEL=(1,1),CLASS=A
/*ROUTE PRINT SYSTEM(USERID)
/*ROUTE PUNCH SYSTEM(USERID)
//*****
//***                                     ***
//***                                     ***
//***                                     ***
//*****
//A      EXEC PGM=CLTP40
//STEPLIB DD DSN=ACPO000.DEVP.TEST.LK,DISP=SHR
//RTL     DD DISP=OLD,UNIT=VTAPE,VOL=SER=A00147,
//         DSN=RTA.TAPE
//PRINT  DD SYSOUT=A,DCB=BLKSIZE=81

```

Figure 34. Sample JCL for the CLAW Process Trace Postprocessor

## CLAW Process Trace Postprocessor

Figure 35 on page 352 provides a sample of process output that is created by the Common Link Access to Workstation (CLAW) process trace postprocessor (CLTP). The process trace postprocessor indicates the system routines in the CCLAW1 CSECT that were entered during system cycle-up.

CLTP reads CLAW process trace output that has been written to the RTA/RTL tape from the ICTRCE trace block and prints it. Each page of the trace output starts with a header followed by variable entry or exit data depending on the presence or absence of trace information associated with the given routine.

The conventions used in the trace are as follows:

- The entry and exit for all routines start with a hyphen (-).
  - routine STRT**                      Routine starts.
  - routine END**                      Routine ends.
- Some codes have been added to the sample trace to help you find the referenced information.

**Code    Description of Information in the Process Trace**

- [A]**    Routine CLXAMAIN starts. Because there are other nested calls in this routine, it is not followed immediately by END.
- [B]**    Routine CLXATRANS starts and the contents of pertinent control blocks or data areas are shown.
- [C]**    Routine CLXTCONN starts and calls routine CLXTSEND.
- [D]**    Routine CLXTSEND starts. No trace data is available for the pertinent control blocks or data areas and routine CLXTSEND ends.
- [D<sub>2</sub>]**   Routine CLXTSEND ends.
- [C<sub>2</sub>]**   Routine CLXTCONN ends.
- [B<sub>2</sub>]**   Routine CLXTRANS ends.
- [A<sub>2</sub>]**   Routine CLXAMAIN ends.



1CLAW PROCESS TRACE	DATE	TIME	OFFLOAD	SDA	PAGE#
0	17SEP	11.10.01	OS2TCP	0240	002
-CLXWVCCC	STRT				
-CLXWVCCC	END				
00(,R0)					
00000000					
-CLXAMAIN	END				
0VCLRC					
00000000					
-CLXAMAIN	STRT				
0VCLTYPE					
03					
-CLXIOI	STRT				
0CLWIOI					
008040070198B0300C0000000241000000000000					
-CLXIOI	END				
00(,R0)					
00000000					
-CLXWTRRT	STRT				
-CLXISUB	STRT				
0VCUFUNC					
07					
-CLXISUB	END				
00(,R0)					
00000000					
-CLXWTRRT	END				
00(,R0)					
00000000					
-CLXISUB	STRT				
0VCUFUNC					
07					
-CLXTRECV	STRT				
-CLXTRECV	END				

Figure 35. Sample Output from the CLAW Process Trace Postprocessor (Part 2 of 11)

1CLAW PROCESS TRACE	DATE	TIME	OFFLOAD	SDA	PAGE#
0	17SEP	11.10.01	OS2TCP	0240	003
00(,R0)					
00000000					
-CLXISUB	END				
00(,R0)					
00000000					
-CLXWVCCC	STRT				
-CLXWVCCC	END				
00(,R0)					
00000000					
-CLXRVCCC	STRT				
-CLXRVCCC	END				
00(,R0)					
00000000					
-CLXAMAIN	END				
0VCLRC					
00000000					
-CLXAMAIN	STRT				
0VCLTYPE					
03					
-CLXIOI	STRT				
0CLWIOI					
008040 I01988068008000100240000001000000					
-CLXIOI	END				
00(,R0)					
00000000					
-CLXAMAIN	END				
0VCLRC					
00000000					
-CLXAMAIN	STRT				
0VCLTYPE					
05					

Figure 35. Sample Output from the CLAW Process Trace Postprocessor (Part 3 of 11)

```

1CLAW PROCESS TRACE      DATE      TIME      OFFLOAD    SDA      PAGE#
0                        17SEP     11.10.01  0S2TCP    0240     004
-CLXPOLL   STRT
0CLPOLL
 8000000000000000
-CLXPOLL   END
0VCLRC
 00000000
-CLXAMAIN  END
0VCLRC
 00000000
-CLXAMAIN  STRT
0VCLTYPE
 03
-CLXIOI    STRT
0CLWIOI
 008040 I019880A8008000100240000001000000
-CLXIOI    END
00(,R0)
 00000000
-CLXISUB   STRT
0VCUFUNC
 07
-CLXTRECV  STRT
-CLXTRECV  END
00(,R0)
 00000000
-CLXISUB   END
00(,R0)
 00000000
-CLXRVCCC  STRT
-CLXRVCCC  END
00(,R0)
 00000000

```

Figure 35. Sample Output from the CLAW Process Trace Postprocessor (Part 4 of 11)



```

1CLAW PROCESS TRACE      DATE      TIME      OFFLOAD    SDA      PAGE#
0                        17SEP     11.10.01  0S2TCP    0240     006
0VCLRC
00000000
-CLXAMAIN  STRT
0VCLTYPE
03
-CLXIOI    STRT
0CLWIOI
008040 I019880 Y008000100240000001000000
-CLXIOI    END
00(,R0)
00000000
-CLXAMAIN  END
0VCLRC
00000000
-CLXAMAIN  STRT
0VCLTYPE
05
-CLXPOLL   STRT
0CLPOLL
8000000000000000
-CLXPOLL   END
0VCLRC
00000000
-CLXAMAIN  END
0VCLRC
00000000
-CLXAMAIN  STRT
0VCLTYPE
06
-CLXPAGR   STRT
-CLXPAGR   END
00(,R0)

```

Figure 35. Sample Output from the CLAW Process Trace Postprocessor (Part 6 of 11)



```

1CLAW PROCESS TRACE      DATE      TIME      OFFLOAD    SDA      PAGE#
0                        17SEP     11.10.01  0S2TCP    0240     008
0CLPOLL
 8000000000000000
-CLXPOLL      END
0VCLRC
 00000000
-CLXAMAIN    END
0VCLRC
 00000000
-CLXAMAIN    STRT
0VCLTYPE
 04
-CLXTRANS    STRT
0CLTRBK
 0000000007020000000400010000000001AA58000037400000000030000000000000000000000
0VCTFUNC
 07
0VCTPATH
 0004
-CLXTSEND    STRT
-CLXTSEND    END
00(,R0)
 00000000
-CLXTRANS    END
0VCTRC
 00000000
-CLXWVCCC    STRT
-CLXWVCCC    END
00(,R0)
 00000000
-CLXAMAIN    END
0VCLRC
 00000000

```

Figure 35. Sample Output from the CLAW Process Trace Postprocessor (Part 8 of 11)

```

1CLAW PROCESS TRACE      DATE      TIME      OFFLOAD   SDA      PAGE#
0                        17SEP     11.10.01  OS2TCP   0240     009
-CLXAMAIN  STRT
0VCLTYPE
 05
-CLXPOLL   STRT
0CLPOLL
8000000000000000
-CLXPOLL   END
0VCLRC
00000000
-CLXAMAIN  END
0VCLRC
00000000
-CLXAMAIN  STRT
0VCLTYPE
 03
-CLXIOI    STRT
0CLWIOI
008040070198B0500C0000000241000000000000
-CLXIOI    END
00(,R0)
00000000
-CLXWTRRT  STRT
-CLXWTRRT  END
00(,R0)
00000000
-CLXWVCCC  STRT
-CLXWVCCC  END
00(,R0)
00000000
-CLXAMAIN  END
0VCLRC
00000000

```

Figure 35. Sample Output from the CLAW Process Trace Postprocessor (Part 9 of 11)





---

## Appendix B. ISO-C Structures Called by Socket API Functions

The parameter lists for some C language socket calls include a pointer to a data structure defined by a C structure. Listed below are some of the C structures called by the socket API functions defined in this publication. These structures are defined in the `socket.h` header file, which needs to be included by any socket application program that uses these structures.

---

### Structures Defined in the `socket.h` Header File

The following structures are used by the `accept`, `bind`, `connect`, `gethostname`, `getpeername`, `getsockname`, `recvfrom`, and `sendto` socket API functions.

- `struct sockaddr`

```
struct sockaddr
{
    unsigned short integer sa_family;    /* address family */
    char sa_data[14];    /* up to 14 bytes of direct address */
};
```
- `struct in_addr`

```
struct in_addr
{
    unsigned long integer s_addr;
};
```
- `struct sockaddr_in`

```
struct sockaddr_in
{
    short    sin_family;
    unsigned short integer sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8];
};
```

---

### Additional Structures

Additional structures defined in the `socket.h` header file are:

- `struct linger`

The `linger` structure is called by the `getsockopt` and `setsockopt` socket API functions.

```
struct linger
{
    int    l_onoff;    /* option on/off */
    int    l_linger;    /* linger time */
};
```

- `struct iovec`

The `iovec` struct is called by the `writew` socket API function.

```
struct iovec
{
    char *iov_base;
    int  iov_len;
}
```



## Appendix C. Socket Error Return Codes

This appendix contains socket error return codes that apply to the socket APIs. See Table 9 for the error codes, message names, and error code descriptions. The table is sorted numerically by error code number.

Table 9. Socket Error Return Codes

Error Codes	Symbolic Name	Description
1	SOCPERM	Not owner
3	SOCSRCH	No such process
4	SOCINTR	Interrupted system call
6	SOCNXIO	No such device or address
9	SOCBADF	Bad file number
13	SOCACCES	Permission denied
14	SOCFAULT	Bad address
22	SOCINVAL	Invalid argument
24	SOCMFILE	Too many open files
32	SOCPIPE	Broken pipe
35	SOCWOULDBLOCK	Operation would block
36	SOCINPROGRESS	Operation now in progress
37	SOCALREADY	Operation already in progress
38	SOCNOTSOCK	Socket operation on non-socket
39	SOCDESTADDRREQ	Destination address required
40	SOCMSGSIZE	Message too long
41	SOCPROTOTYPE	Protocol wrong type for socket
42	SOCNOPROTOOPT	Protocol not available
43	SOCPROTONOSUPPORT	Protocol not supported
44	SOC SOCKTNOSUPPORT	Socket type not supported
45	SOCOPNOTSUP	Operation not supported on socket
47	SOC AFNOSUPPORT	Address family not support by protocol family
48	SOCADDRINUSE	Address already in use
49	SOCADDRNOTAVAIL	Can't assign requested address
50	SOCNETDOWN	Network is down
51	SOCNETUNREACH	Network is unreachable
52	SOCNETRESET	Network is dropped connection on reset
53	SOC CONNABORTED	Software caused connection abort
54	SOC CONNRESET	Connection reset by peer
55	SOCNOBUFS	No buffer space available
56	SOCISCONN	Socket is already connected
57	SOCNOTCONN	Socket is not connected
58	SOC SHUTDOWN	Can't send after socket shutdown
59	SOC TOOMANYREFS	Too many references: can't splice
60	SOC TIMEDOUT	Connection timed out

Table 9. Socket Error Return Codes (continued)

Error Codes	Symbolic Name	Description
61	SOCONNREFUSED	Connection refused
62	SOCLOOP	Too many levels of symbolic links
63	SOCNAMETOOLONG	File name too long
64	SOCHOSTDOWN	Host is down
65	SOCHOSTUNREACH	No route to host
66	SOCNOTEMPTY	Directory not empty
100	SOCOS2ERR	OS/2 Error
251	EINACTWS	Offload device deactivated
252	E1052STATE	System cycling to 1052 state
253	EINACT	All offload devices deactivated
254	ESYSTEMERROR	Socket system error
1000	OFFLOADTIMEOUT	Offload device time-out
1004	EIBMIUCVERR	Failure with offload device
2000	SOCFDNOTFOUND	FD entry cannot be found
2001	SOCIPNOTFOUND	IP entry cannot be found

---

## Appendix D. Sample Application Driver Code

This appendix lists sample socket drivers to use with TCP/IP support. Use these drivers as sample code when you write your own socket applications. The drivers are:

- activate\_on\_receipt Transmission Control Protocol (TCP) server
- activate\_on\_receipt Transmission Control Protocol (TCP) child
- Transmission Control Protocol (TCP) server
- Transmission Control Protocol (TCP) client
- User Datagram Protocol (UDP) server
- User Datagram Protocol (UDP) client.

---

### activate\_on\_receipt Transmission Control Protocol (TCP) Server

```

/*****
/* This is a stream socket server sample program that
/* accepts a connect request from a client program
/* and issues an activate_on_receipt socket API call to
/* create a child program. When the message from the
/* client program is received, the child program is
/* activated with a new ECB.
*****/

/*****
/* TPF Standard Header Files
*****/
#include <tpfeq.h>
#include <tpfio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <types.h>
#include <socket.h>          /* Socket API Header File */

#define return_error      -1

qxye()
{
    struct sockaddr_in server_sockaddr_in, name_sockaddr_in;

    char server_name[50];

    int client_sock;
    int rc;
    int server_sock;
    int sockaddr_in_length;
    int num_of_clients;
    unsigned char aorparm[8];
    unsigned char aorpgm[4]= {'Q','X','Y','K'};

    sockaddr_in_length = sizeof(struct sockaddr_in);

    /*****
    /* Create a stream socket */
    *****/
    server_sock = socket(AF_INET,SOCK_STREAM,0);
    if (server_sock == return_error)
    {
        printf("Error in opening stream socket\n");
        exit(0);
    }

```

```

}

/*****/
/* Bind it */
/*****/
server_sockaddr_in.sin_family      = AF_INET;
server_sockaddr_in.sin_addr.s_addr = INADDR_ANY;
server_sockaddr_in.sin_port       = 5003;
rc = bind(server_sock, (struct sockaddr *)&server_sockaddr_in,
           sockaddr_in_length);
if (rc == return_error)
{
    printf("Error in binding stream - %d\n", sock_errno());
    close(server_sock);
    exit(0);
}

/*****/
/* Listen to any socket clients.*/
/*****/
rc = listen(server_sock, 5);
if (rc == return_error)
{
    printf("Error in listening client socket - %d\n", sock_errno());
    close(server_sock);
    exit(0);
}

/*****/
/* Call gethostname to obtain the server name.*/
/*****/
rc = gethostname(server_name, sizeof(server_name));
if (rc == return_error)
{
    printf("Error in getting host name - %d\n", sock_errno());
}

/*****/
/* Print out server information.*/
/*****/
printf("\nServer Information\n");
printf("-----\n");
printf("Socket      = %d\n", server_sock);
printf("Host ID      = %x\n", gethostid());
printf("Host Name     = %s\n", server_name);
printf("Host Family   = %x\n", name_sockaddr_in.sin_family);
printf("Host Address  = %x\n", name_sockaddr_in.sin_addr.s_addr);
printf("Host Port No  = %d\n", name_sockaddr_in.sin_port);
printf("-----\n\n");

num_of_clients = 0;

/*****/
/* Accept a socket client. */
/*****/
client_sock = accept(server_sock, (struct sockaddr *)&name_sockaddr_in,
                    &sockaddr_in_length);
if (client_sock == return_error)
{
    printf("Cannot accept any new clients - %d\n", sock_errno());
    close(server_sock);
    exit(0);
}
else
{
    (void)memcpy(aorparm, &server_sock, sizeof(server_sock));
    (void)memcpy(aorparm+sizeof(server_sock), &client_sock,

```

```

        sizeof(client_sock));
rc = activate_on_receipt(client_sock,aorparm,aorpgm);
if (rc == -1)
{
    printf("error in activate_on_receipt - %d\n",sock_errno());
    close(server_sock);
    exit(0);
}
else
{
    num_of_clients = num_of_clients + 1;
}
}

rc = close(server_sock);
if (num_of_clients == 1)
    printf("%d Client was activated\n\n", num_of_clients);
else
    printf("%d Clients were activated\n\n", num_of_clients);
printf("Terminated communication normally\n");

exit(0);
}

```

---

## activate\_on\_receipt Transmission Control Protocol (TCP) Child Server

```

/*****
/* This is a stream child server sample program which is activated*/
/* when a message is received from the client program.          */
*****/

/*****
/* TPF Standard Header Files                                     */
*****/

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

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

#include <socket.h>          /* Socket API Header File */
#include <types.h>

#define PRIMECRAS           0x010000

#define return_error       -1

#define MAX_BUFFER_LEN     32 * 1024

qxyk()
{
    struct sockaddr_in name_sockaddr_in;

    char *read_addr;

    int client_sock;
    int optlen;
    int optval;
    int rc;
    int read_len;
    int sockaddr_in_length;
    int echo_len;

    sockaddr_in_length = sizeof(struct sockaddr_in);

```

```

(void)memcpy(&read_addr,&(ecbptr()->ebw012),
            sizeof(read_addr));
(void)memcpy(&read_len, &(ecbptr()->ebw016),sizeof(read_len));

client_sock = (int)ecbptr()->ebrout;
ecbptr()->ebrout = PRIMECRAS;

optval = MAX_BUFFER_LEN;
optlen = sizeof(optval);
rc = setsockopt(client_sock,SOL_SOCKET,SO_SNDBUF,(char *)&optval,
               optlen);

/*****
/* Call getpeername() to obtain client's information.*/
*****/
rc = getpeername(client_sock,(struct sockaddr *)&name_sockaddr_in,
                &sockaddr_in_length);

if (rc == return_error)
{
    printf("Error in getting peer name\n",sock_errno());
    close(client_sock);
    exit(0);
}

/*****
/* Print out client information.*/
*****/
printf("Client Information\n");
printf("-----\n");
printf("Socket          = %d\n",client_sock);
printf("Client Address   = %x\n",name_sockaddr_in.sin_addr.s_addr);
printf("Client Port No   = %d\n",name_sockaddr_in.sin_port);
printf("-----\n");

/*****
/* Read client's message. */
*****/
echo_len = read(client_sock,read_addr,read_len);
if (echo_len == return_error)
{
    printf("Error in reading a msg - %d\n",sock_errno());
    close(client_sock);
    exit(0);
}
else
{
    printf("messages = %s\n",read_addr);
}
rc = write(client_sock,read_addr,echo_len);
if (rc == return_error)
{
    printf("Error in sending echo - %d\n",sock_errno());
    (void)close(client_sock);
    exit(0);
}

/*****
/* Shutdown the communication link between client and server.*/
*****/
rc = shutdown(client_sock,2);
if (rc == return_error)
{
    printf("Error in shutting down the client - %d\n",
          sock_errno());
}

```

```

/*****/
/* Clean up all sockets and exit the program.*/
/*****/
rc = close(client_sock);
if (rc == return_error)
{
    printf("Error in closing client socket - %d\n",
          sock_errno());
}
printf("Terminated communication normally-stream AOR child\n");
exit(0);
}

```

---

## Transmission Control Protocol (TCP) Server

```

/*****/
/* This is a stream socket server sample program that */
/* accepts a connect request from a client program. */
/* When a message from the client program is received, */
/* the TCP server program echoes back the message to the */
/* client program. */
/*****/

/*****/
/* TPF Standard Header Files */
/*****/
#include <tpfeq.h>
#include <tpfio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <socket.h>          /* Socket API Header File */

#define return_error      -1

#define MAX_BUFFER_LEN    1024

qxyd()
{
    struct sockaddr_in server_sockaddr_in, name_sockaddr_in;

    char *recv_client_msg;
    char server_name[50];

    int client_sock;
    int echo_len;
    int rc;
    int server_sock;
    int sockaddr_in_length;

    recv_client_msg = malloc(MAX_BUFFER_LEN);
    sockaddr_in_length = sizeof(struct sockaddr_in);

    /*****/
    /* Create a stream socket.*/
    /*****/
    server_sock = socket(AF_INET,SOCK_STREAM,0);
    if (server_sock == return_error)
    {
        printf("Error in opening stream socket\n");
        exit(0);
    }

    /*****/

```

```

/* Bind it.*/
/*****/
server_sockaddr_in.sin_family      = AF_INET;
server_sockaddr_in.sin_addr.s_addr = INADDR_ANY;
server_sockaddr_in.sin_port       = 5003;
rc = bind(server_sock,(struct sockaddr *)&server_sockaddr_in,
          sockaddr_in_length);
if (rc == return_error)
{
    printf("Error in binding stream - %d\n",sock_errno());
    close(server_sock);
    exit(0);
}

/*****/
/* Listen to any socket clients.*/
/*****/
rc = listen(server_sock,5);
if (rc == return_error)
{
    printf("Error in listening client socket - %d\n",sock_errno());
    close(server_sock);
    exit(0);
}

/*****/
/* Call gethostname to obtain the server name */
/*****/
rc = gethostname(server_name,sizeof(server_name));
if (rc == return_error)
{
    printf("Error in getting host name - %d\n",sock_errno());
}

/*****/
/* Print out server information.*/
/*****/
printf("\nServer Information\n");
printf("-----\n");
printf("Socket          = %d\n",server_sock);
printf("Host ID          = %x\n",gethostid());
printf("Host Name         = %s\n",server_name);
printf("Host Family       = %x\n",name_sockaddr_in.sin_family);
printf("Host Address      = %x\n",name_sockaddr_in.sin_addr.s_addr);
printf("Host Port No      = %d\n",name_sockaddr_in.sin_port);
printf("-----\n\n");

/*****/
/* Accept a socket client. */
/*****/
client_sock = accept(server_sock,(struct sockaddr *)&name_sockaddr_in,
                    &sockaddr_in_length);
if (client_sock == return_error)
{
    printf("Cannot accept any new clients - %d\n",sock_errno());
    close(server_sock);
    exit(0);
}

/*****/
/* Print out client information.*/
/*****/
printf("Client Information\n");
printf("-----\n");
printf("Socket              = %d\n",client_sock);
printf("Client Family       = %x\n",name_sockaddr_in.sin_family);
printf("Client Address      = %x\n",name_sockaddr_in.sin_addr.s_addr);

```

```

printf("Client Port No    = %d\n",name_sockaddr_in.sin_port);
printf("-----\n");

/*****
/* Read client's message to find out how many messages the*
/* server is receiving.          */
*****/
echo_len = read(client_sock,recv_client_msg,MAX_BUFFER_LEN);
if (echo_len == return_error)
{
    printf("Error in reading message - %d\n",sock_errno());
    (void)close(client_sock);
    (void)close(server_sock);
    exit(0);
}
else
{
    printf("messages = %s\n",recv_client_msg);
}

rc = write(client_sock,recv_client_msg,echo_len);
if (rc == return_error)
{
    printf("Error in sending echo - %d\n",sock_errno());
    (void)close(client_sock);
    (void)close(server_sock);
    exit(0);
}

/*****
/* Shutdown the communication link between client and server.*
*****/
rc = shutdown(client_sock,2);
if (rc == return_error)
{
    printf("Error in shutting down the client - %d\n",
        sock_errno());
}

/*****
/* Clean up all sockets and exit the program.*
*****/
rc = close(client_sock);
if (rc == return_error)
{
    printf("Error in closing client socket - %d\n",sock_errno());
}

(void)close(server_sock);
printf("Terminated communication normally\n");
exit(0);
}

```

---

## Transmission Control Protocol (TCP) Client

```

/*****
/* This is a stream socket client sample program that sends a    */
/* connect socket API request to the server program to bring up  */
/* the connection between the server and client. The TCP server */
/* also sends a message and receives an echo from the server.   */
*****/

#include <tpfeq.h>
#include <tpfio.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

```

```

#include <types.h>
#include <socket.h>

#define MAX_BUFFER_LEN      1024
#define NUM_PARAMETER      4
#define SEND_MSG_LEN       5

#define return_ok           0
#define return_error       -1

#define read_buffer         0
#define use_previous_buffer 1

qxyh(char *argv[], u_short num_arg)
{
    struct sockaddr_in client_sockaddr_in;
    struct sockaddr_in server_sockaddr_in;

    char *recv_client_message;
    char *send_client_message;

    short dest_portnum;

    int client_sock;
    int dest_address;
    int rc;
    int sockaddr_in_length;

    /******
    /* Check the argument count.
    /******

    if (num_arg != NUM_PARAMETER)
    {
        printf("Invalid parameters :\n");
        printf("ztest sock client stream block <server IP addr> \
            <server port no.> <no. of msg> <msg size>\n");
        exit(0);
    }

    dest_address = inet_addr(argv[0]);
    dest_portnum = atoi(argv[1]);

    /******
    /* Print out the client's parameter list.*/
    /******
    printf("Parameter List :\n");
    printf("-----\n");
    printf("Dest address      = %x\n",dest_address);
    printf("Dest portnum       = %d\n",dest_portnum);
    printf("-----\n\n");

    /******
    /* Initialize input/output buffer size.*/
    /******
    send_client_message = (char *)malloc(MAX_BUFFER_LEN);
    recv_client_message = (char *)malloc(MAX_BUFFER_LEN);
    sockaddr_in_length = sizeof(struct sockaddr_in);

    /******
    /* Create a stream socket.*/
    /******
    client_sock = socket(AF_INET,SOCK_STREAM,0);
    if (client_sock == return_error)
    {
        printf("Error in opening a stream socket\n");
        return(return_error);
    }

```

```

}

/*****
/* Bind a local name to the socket. When using the IP routing */
/* table, this bind API should be skipped. */
*****/
client_sockaddr_in.sin_family = AF_INET;
client_sockaddr_in.sin_port = 0;
client_sockaddr_in.sin_addr.s_addr = 0;
rc = bind(client_sock,(struct sockaddr *)&client_sockaddr_in,
          sockaddr_in_length);
if (rc == return_error)
{
    printf("Error in binding - %d\n",sock_errno());
    rc = close(client_sock);
    exit(0);
}

/*****
/* Print out client information.*/
*****/
printf("Client Information\n");
printf("-----\n");
printf("Client sock = %d\n",client_sock);
rc = getsockname(client_sock,(struct sockaddr *)&client_sockaddr_in,
                 &sockaddr_in_length);
if (rc == return_ok)
{
    printf("Client Address =%x\n",client_sockaddr_in.sin_addr.s_addr);
    printf("Client Port No = %d\n",client_sockaddr_in.sin_port);
}
printf("-----\n\n");

/*****
/* Connect to a connection-oriented server.*/
*****/
server_sockaddr_in.sin_family = AF_INET;
server_sockaddr_in.sin_port = dest_portnum;
server_sockaddr_in.sin_addr.s_addr = dest_address;
rc = connect(client_sock,(struct sockaddr *)&server_sockaddr_in,
             sockaddr_in_length);
if (rc == return_error)
{
    printf("Error in connecting - %d\n",sock_errno());
    rc = close(client_sock);
    exit(0);
}

printf("Server Information\n");
printf("-----\n");
printf("Server Address = %x\n",server_sockaddr_in.sin_addr.s_addr);
printf("Server Port No = %d\n",server_sockaddr_in.sin_port);
printf("-----\n\n");

/*****
/* Send the server a message.*/
*****/
(void)memset(send_client_message,'a',SEND_MSG_LEN - 1);
*(send_client_message+SEND_MSG_LEN) = 0;
rc = send(client_sock,send_client_message,SEND_MSG_LEN,0);
if (rc == return_error)
{
    printf("Error in sending message - %d\n",sock_errno());
    rc = close(client_sock);
    exit(0);
}
else

```

```

        printf("send msg = %s\n",send_client_message);

/*****
/* Receive an echo message from the server.          */
*****/
rc = recv(client_sock,recv_client_message,MAX_BUFFER_LEN,0);
if (rc == return_error)
{
    printf("Error in receiving message - %d\n",sock_errno());
    rc = close(client_sock);
    exit(0);
}
else
    printf("recv msg = %s\n",recv_client_message);

/*****
/* Clean up the connection and exit */
*****/
rc = shutdown(client_sock,2);
if (rc == return_error)
{
    printf("Error in shutting down - %d\n",sock_errno());
}

rc = close(client_sock);
if (rc == return_error)
{
    printf("Error in closing the socket - %d\n",sock_errno());
    exit(0);
}

printf("Terminated communication normally-stream client\n");
exit(0);
}

```

---

## User Datagram Protocol (UDP) Server

```

/*****
/* This is a datagram socket server sample program.    */
/* When a message from the client program is received, */
/* it echoes back the message to the client program.    */
*****/

/*****
/* TPF Standard Header Files                          */
*****/
#include <tpfeg.h>
#include <tpfio.h>

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

#include <types.h>
#include <socket.h>          /* Socket API Header File */

#define return_error      -1

qxyb()
{
    const message_size = 32 * 1024;

    struct sockaddr_in client_sockaddr_in;
    struct sockaddr_in server_sockaddr_in;

    char *recv_client_message;

```

```

char *send_client_message;

int echo_len;
int optlen;
int optval;
int rc;
int server_sock;
int sockaddr_in_length;

send_client_message = malloc(message_size);
recv_client_message = malloc(message_size);
sockaddr_in_length = sizeof(struct sockaddr_in);

/*****
/* Create a datagram socket.*/
*****/
server_sock = socket(AF_INET,SOCK_DGRAM,0);
if (server_sock == return_error)
{
    printf("Error in opening a datagram socket\n");
    exit(0);
}
printf("Host sock = %d\n",server_sock);

/*****
/* Bind a local name to the socket.*/
*****/
server_sockaddr_in.sin_family      = AF_INET;
server_sockaddr_in.sin_port        = 5001;
server_sockaddr_in.sin_addr.s_addr = INADDR_ANY;
rc = bind(server_sock,(struct sockaddr *)&server_sockaddr_in,
          sockaddr_in_length);
if (rc == return_error)
{
    printf("Error in binding - %d\n",sock_errno());
    (void)close(server_sock);
    exit(0);
}

/*****
/* Get the server information, and print its data out.*/
*****/
rc = getsockname(server_sock,(struct sockaddr *)&server_sockaddr_in,
                 &sockaddr_in_length);
if (rc != return_error)
{
    printf("Server Information\n");
    printf("-----\n");
    printf("Server sock      - %d\n",server_sock);
    printf("Server IP address - %x\n",
          server_sockaddr_in.sin_addr.s_addr);
    printf("Server port #    - %d\n\n",
          server_sockaddr_in.sin_port);
}

optval = message_size;
optlen = sizeof(optval);
rc = setsockopt(server_sock,SOL_SOCKET,SO_SNDBUF,(char *)&optval,
               optlen);
if (rc == return_error)
{
    printf("Error in setsockopt - %d\n",sock_errno());
    (void)close(server_sock);
    exit(0);
}

```

```

printf(".... Ready for clients ....\n");

/******/
/* Loop forever */
/******/
for (;;)
{
/******/
/* Monitor incoming buffers. */
/******/
rc = recvfrom(server_sock,recv_client_message,message_size,0,
              (struct sockaddr *)&client_sockaddr_in,
              &sockaddr_in_length);
if (rc == return_error)
{
printf("Error in receiving message - %d\n",sock_errno());
break;
}
else
{
/******/
/* Print out the client information and message length*/
/* and then echo the same message back to that client.*/
/******/
echo_len = rc;
(void)memcpy(send_client_message,recv_client_message,echo_len);
rc = sendto(server_sock,send_client_message,echo_len,0,
            (struct sockaddr *)&client_sockaddr_in,
            sockaddr_in_length);
if (rc == return_error)
printf("Error in sending message to a client - %d\n",
      sock_errno());
}
} /* end of for(;;) */

/******/
/* Return the socket back to the system and exit normally. */
/******/
rc = close(server_sock);
if (rc == return_error)
{
printf("Error in closing the socket - %d\n",sock_errno());
printf("DATAGRAM NOAOR server terminated\n");
exit(0);
}

printf("Datagram server terminated\n");
exit(0);
}

```

---

## User Datagram Protocol (UDP) Client

```

/******/
/* This is a datagram socket client sample program which sends a */
/* request to the server program and receives an echo message */
/* back from the server. */
/******/
#include <tpfeq.h>
#include <tpfio.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include <types.h>
#include <socket.h>

qxyg(char *argv[], u_short num_arg)

```

```

{
#define NUM_PARAMETER      5

#define return_ok          0
#define return_error      -1

#define no                 0
#define yes                1

    /******
    /* This is the maximum message size that will be supported */
    /******
    const max_msg_size = 32 * 1024;

    struct sockaddr_in client_sockaddr_in; /* Client internet addr */
    struct sockaddr_in server_sockaddr_in; /* Server internet addr */

    char msg = '0';
    char *recv_client_message;
    char *send_client_message;

    int client_sock;
    int msg_lost;
    int msg_rcv;
    int msg_resend;
    int msg_resyn;
    int rc;
    int redo_select;
    int sockaddr_in_length;
    int sock_list[1];
    /******
    /* Argument List */
    /******
    int dest_ip;
    int dest_port;
    int num_msg;
    int siz_msg;
    int num_rsend;

    /******
    /* Check the input argument list. If the number of arguments is */
    /* incorrect, print out the help message. */
    /******

    if (num_arg != NUM_PARAMETER)
    {
        printf("Invalid parameter(s) :\n");
        printf("ztest sock client datagram block <IP addr> <port #> \
                <no. of msg> <msg size> <resend>\n");
        printf("ip addr      - Server's IP address\n");
        printf("port #        - Server's port number\n");
        printf("num of msg   - Number of messages to be sent to \
                the server\n");
        printf("msg size     - Size of each message to be sent\n");
        printf("resend      - Number of times to resend the lost message \
                \n\n");

        exit(0);
    }
    else
    {
        /******
        /* Extract the IP address, port number, number of messages, */
        /* size of each message, and number of resends. */
        /******
        dest_ip = inet_addr(argv[0]);
        dest_port = atoi(argv[1]);

```

```

    num_msg = atoi(argv[2]);
    siz_msg = atoi(argv[3]);
    num_rsend = atoi(argv[4]);
}
if (siz_msg < 0)
    printf("qxyg: %d is an invalid message size\n",siz_msg);

/*****
/* Create send/receive buffer based on user defined buffer size.*/
*****/
send_client_message = (char *)malloc(max_msg_size);
recv_client_message = (char *)malloc(max_msg_size);

msg_resend = 0;
msg_resyn = 0;
msg_lost = 0;
sockaddr_in_length = sizeof(struct sockaddr_in);

/*****
/* Create a datagram socket.*/
*****/
client_sock = socket(AF_INET,SOCK_DGRAM,17);
if (client_sock == return_error)
{
    printf("qxyg: Error in opening a datagram socket\n");
    exit(0);
}

/*****
/* Bind a local name to the socket.*/
*****/
client_sockaddr_in.sin_family = AF_INET;
client_sockaddr_in.sin_port = 0;
client_sockaddr_in.sin_addr.s_addr = 0;
rc = bind(client_sock,(struct sockaddr *)&client_sockaddr_in,
sockaddr_in_length);
if (rc == return_error)
{
    printf("qxyg: Error in binding - %d\n",sock_errno());
    (void)close(client_sock);
    exit(0);
}

/*****
/* Set socket to send/receive more than max_buffer_size.*/
*****/
rc = setsockopt(client_sock,SOL_SOCKET,SO_SNDBUF,
(char *)&max_msg_size,sizeof(max_msg_size));
if (rc != return_ok)
{
    printf("qxyg: Error in setsockopt-SO_SNDBUF-%d\n",sock_errno());
    (void)close(client_sock);
    exit(0);
}

/*****
/* Define server's internet address which is used for sendto() and */
/* recvfrom(). */
*****/
server_sockaddr_in.sin_family = AF_INET;
server_sockaddr_in.sin_port = dest_port;
server_sockaddr_in.sin_addr.s_addr = dest_ip;

rc = connect(client_sock,(struct sockaddr *)&server_sockaddr_in,
sockaddr_in_length);
if (rc == return_error)
{

```

```

printf("qxyg: Error in connecting - %d\n",sock_errno());
(void)close(client_sock);
exit(0);
}

/*****
/* Get the client information, and print client/server data out. */
*****/
rc = getsockname(client_sock,(struct sockaddr *)&client_sockaddr_in,
&sockaddr_in_length);
if (rc != return_error)
{
printf("Client Information\n");
printf("-----\n");
printf("Client sock      - %d\n",client_sock);
printf("Client IP address - %x\n",          \
      client_sockaddr_in.sin_addr.s_addr);
printf("Client port #    - %d\n",client_sockaddr_in.sin_port);
printf("Server IP address - %x\n",          \
      server_sockaddr_in.sin_addr.s_addr);
printf("Server port #    - %d\n\n",server_sockaddr_in.sin_port);
}

/*****
/* Set up the first message to send to the server.*/
*****/
(void)memset(send_client_message,msg,siz_msg);
send_client_message[siz_msg - 1] = 0;

/*****
/* Set up the sock_list for select() - use it for testing time out.*/
*****/
sock_list[0] = client_sock;

/*****
/* Loop until all messages are sent.*/
*****/
for (msg_rcv = 1; msg_rcv <= num_msg;)
{

/*****
/* Send a message to the server.*/
*****/

rc = send(client_sock,send_client_message,siz_msg,0);
if (rc == return_error)
{
/*****
/* If an error occurred, resend the message. */
*****/
printf("qxyg: Error in sending message to server-try again- \
      %d\n",sock_errno());
msg_resend++;
}
else
{
/*****
/* A message was successfully sent, so monitor the read. */
/* If a message arrives within 15 seconds, read it in. */
/* Otherwise, a message was lost, so send it again. */
*****/
redo_select = yes;
for (;redo_select; )
{
rc = select(sock_list,1,0,0,15000);
if (rc >= 1)

```

```

{
/*****
/* Read the message. If there is a problem, resend again.*/
/*****
rc = read(client_sock,recv_client_message,max_msg_size);
if (rc == return_error)
{
printf("qxyg : Error in receiving message from server - \
%d\n", sock_errno());
msg_resend++;
}
else
{
/*****
/* If the received message is correct, then process the */
/* message. Otherwise, throw the message out and wait */
/* for the next incoming message. */
/*****
if (*recv_client_message == msg)
{
printf("Recv Msg %d - len = %d Source address = %x\n",
msg_rcv,rc,server_sockaddr_in.sin_addr.s_addr);
msg_rcv++;
msg_lost += msg_resend;
msg_resend = 0;
redo_select = no;

/*****
/* Set up the next message to send.*/
/*****
msg ++;
if (msg > '9')
{
msg = '0';
}
(void)memset(send_client_message,msg,siz_msg);
send_client_message[siz_msg - 1] = 0;
}
else
{
printf("qxyg: Received unexpected data. 1st character = \
%c\n", *recv_client_message);
msg_resyn++;
}
}
}
else
{
/*****
/* If a time-out or an error in select() occurred, resend */
/* the message. */
/*****
msg_resend++;
redo_select = no;
if (rc == 0)
printf("qxyg: Possible lost message# %d .... resend\n",
msg_rcv);
else
{
printf("qxyg: error in select() - %d\n",sock_errno());
printf("Server IP address - %x\n", \
server_sockaddr_in.sin_addr.s_addr);
printf("Server port # - %d\n\n",
server_sockaddr_in.sin_port);
printf("Client terminates abnormally\n");
rc = close(client_sock);
if (rc == return_error)

```

```

        {
            printf("qxyg: Error in closing the socket - %d\n",
                sock_errno());
        }
        exit(0);
    }
    } /* end of else (select) */
} /* end of for (;redo_select;) */
} /* end of else (sendto) */

/*****
/* If the number of times we resent the message exceeds the
/* resend count, terminate this program abnormally.
*****/
if (msg_resend >= num_rsend)
{
    printf("\n");
    printf("qxyg: MSG %d has been resent %d times without any \
        response from the server.\n",msg_rcv,msg_resend);
    printf("This could be caused by busy server or unreachable IP add\
        ress or port number.\n");
    printf("Server IP address - %x\n", \
        server_sockaddr_in.sin_addr.s_addr);
    printf("Server port # - %d\n\n", \
        server_sockaddr_in.sin_port);
    printf("Client terminates abnormally\n");
    (void)close(client_sock);
    exit(0);
}
}

/*****
/* Print out the status. Close down socket and exit this program
*****/
msg_rcv--;
printf("\n");
printf("qxyg: MSG Send/Recv = %d, MSG Resyn = %d, Total MSG Lost = \
    %d\n\n", msg_rcv,msg_resyn,msg_lost-msg_resyn);

rc = close(client_sock);
if (rc == return_error)
{
    printf("qxyg: Error in closing the socket - %d\n",sock_errno());
    exit(0);
}
printf("Client terminates normally\n");
printf("Server IP address - %x\n", \
    server_sockaddr_in.sin_addr.s_addr);
printf("Server port # - %d\n\n",server_sockaddr_in.sin_port);
exit(0);
}

```



---

## Appendix E. TCP/IP Restricted CLAW C Functions: Reference

This appendix contains the restricted Common Link Access to Workstation (CLAW) C functions. The TPF system provides these CLAW C functions to communicate to the IBM 3172 Model 3 Interconnect Controller, or similar workstation, to provide a Transmission Control Protocol/Internet Protocol (TCP/IP) environment. These functions are restricted to system use only. If applications use these restricted functions, the results cannot be predicted.

---

## claw\_accept — Accept a CONNECT Request from the Workstation

### ISO-C only

The `claw_accept` function is not available in the TARGET(TPF) C library.

The `claw_accept` function completes the construction of a logical link that was started by a CONNECT request from the workstation.

## Format

```
#include <claw.h>
int      claw_accept(unsigned int adapter_id,
                    unsigned int path_id,
                    const char *disconn_ep,
                    const char *msg_ep,
                    enum s_rcv_method rcv_method);
```

### adapter\_id

A 4-byte field that contains the adapter ID for this adapter that was returned from the `claw_openadapter` function request.

### path\_id

A 4-byte field that contains the path ID, which indicates the same path over which a connection request was received from a CLAW workstation.

### disconn\_ep

A pointer to a 4-byte field that contains the character string CLA2. This entry point is called asynchronously if the workstation starts a disconnect request to the host. This routine is defined as a TPF real-time program.

### msg\_ep

A pointer to a 4-byte field that contains the character string CLA4. This user exit, which is a nonsocket message user exit, is called asynchronously whenever a message is sent on this path from the workstation to the host, if the path ID is defined with a **rcv\_method** of **PAGERECV**. This routine is defined as a TPF real-time program.

### rcv\_method

A variable specifies how messages are to be received on this path. This variable must belong to the enumeration type `s_rcv_method` defined in `claw.h`. **PAGERECV** is the only **rcv\_method** supported.

#### **PAGERECV**

CLAW calls **msg\_ep** with the address and length of the data that was just received. The application does not need to issue a RECEIVE to receive the data.

#### **SYNCRECV\_FLUSH**

Reserved for future IBM use.

#### **SYNCRECV\_HOLD**

Reserved for future IBM use.

#### **ASYNCRECV\_FLUSH**

Reserved for future IBM use.

#### **ASYNCRECV\_HOLD**

Reserved for future IBM use.

#### **AUTORECV**

Reserved for future IBM use.

## Normal Return

Return code 0 indicates that the function was successful.

## Error Return

Following is a list of return conditions that are returned to programs that call `claw_accept`. See “CLAW Return Codes” on page 405 for a complete list of the return codes and their definitions.

```
RC_CLAW_INVALID_FUNCTION
RC_CLAW_NOT_INITED
RC_CLAW_ADAPTER_NOT_OPEN
RC_CLAW_ACQUIRE_ERROR
RC_CLAW_PATH_NOT_THERE
RC_CLAW_BAD_RECV_METHOD
```

## Programming Considerations

- If the program is not ready to accept an incoming CONNECT, issue `claw_disconnect` instead of `claw_accept`.
- Activating **disconn\_ep** indicates that a DISCONNECT request is issued from the workstation to the host. No corresponding DISCONNECT request to a CLAW workstation is required. If any SEND or RECEIVE requests are active on the logical link, they end with a return code indicating a nonexistent path. A new ECB is created and the parameter list is passed to the program starting at EBW000.
- Activating **msg\_ep**, which is a nonsocket message user exit (CLA4), indicates an incoming CLAW message from the workstation for the paths using the **recv\_method** of **PAGERECV**. A new ECB is created and the parameter list is passed to the program starting at EBW000. See *TPF System Installation Support Reference* for additional information about CLA4, including the length and address of the data.

## Examples

The following example issues the `claw_accept` function to accept a CONNECT request from the workstation.

```
#include <claw.h>

unsigned int  adapter_id;
unsigned int  path_id;
char         disconn_ep[5] = "CLA2";
char         msg_ep[5] = "CLA4";
int          claw_rc;

/* Set up adapter_id with the value returned from the
claw_openadapter and path_id with the value returned from the
claw_connect */
:
:
claw_rc = claw_accept(adapter_id,path_id,disconn_ep,msg_ep,PAGERECV);
/*normal processing path */
:
:
```

## Related Information

- “`claw_openadapter` — Initialize an Adapter” on page 397
- “`claw_connect` — Initiate a Request to Open a Logical Link” on page 390
- “`claw_disconnect` — Remove a Logical Link from an Adapter” on page 393.

---

## claw\_closeadapter — Terminate CLAW Activity on Subchannel Pair

**ISO-C only**

The `claw_closeadapter` function is not available in the TARGET(TPF) C library.

The `claw_closeadapter` function ends all CLAW activity on a given subchannel pair.

### Format

```
#include <claw.h>
int      claw_closeadapter(unsigned int adapter_id);
```

**adapter\_id**

A 4-byte field that contains the adapter ID for the adapter that was returned from the `claw_openadapter` request.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

Following is a list of return conditions that can be returned to the program that calls the `claw_closeadapter` function. See “CLAW Return Codes” on page 405 for a complete list of the return codes and their definitions.

```
RC_CLAW_INVALID_FUNCTION
RC_CLAW_NOT_INITED
RC_CLAW_ADAPTER_NOT_OPEN
RC_CLAW_ACQUIRE_ERROR
RC_CLAW_INIT_ERROR
RC_CLAW_CLOSEDEVICE_ERROR
```

### Programming Considerations

All logical links are broken and the CLAW device no longer monitors the subchannel addresses for activity when the `claw_closeadapter` function is issued. If there are any active logical links open on the subchannel pair, all active SEND and RECEIVE requests end with an error return code and the logical links are disconnected. The appropriate `disconn_ep`, CLA2, is called for each logical link.

### Examples

The following example issues the `claw_closeadapter` function to close the adapter.

```
#include <claw.h>

unsigned int      adapter_id;
int              claw_rc;

/* Set up adapter_id with the value returned from the
claw_openadapter function */
.
claw_rc = claw_closeadapter(adapter_id);
```

```
        /*normal processing path */  
        :
```

## **Related Information**

- “claw\_openadapter — Initialize an Adapter” on page 397
- “claw\_end — Terminate All CLAW Activity” on page 395.

---

## claw\_connect — Initiate a Request to Open a Logical Link

### ISO-C only

The `claw_connect` function is not available in the TARGET(TPF) C library.

The `claw_connect` function starts a request to open a logical link on an active CLAW adapter.

## Format

```
#include <claw.h>
int      claw_connect(unsigned int adapter_id,
                    unsigned int *path_id,
                    const char *disconn_ep,
                    const char *msg_ep,
                    enum s_rcv_method rcv_method,
                    void *path_anchor,
                    const char *hostappl,
                    const char *wsappl);
```

### adapter\_id

A 4-byte field that contains the adapter ID for the adapter that was returned from the `claw_openadapter` request.

### path\_id

A pointer to a 4-byte field that contains the unique value that will be filled in by the CLAW workstation to identify this path. This **path\_id** value must be used on subsequent `claw_send` and `claw_disconnect` requests for this path.

### disconn\_ep

A pointer to a 4-byte field that contains the character string CLA2. This entry point is called asynchronously if the workstation starts a disconnect request to the host. This routine is defined as a TPF real-time program.

### msg\_ep

A pointer to a 4-byte field that contains the character string CLA4. This user exit, which is a nonsocket message user exit, is called asynchronously whenever a message is sent on this path from the workstation to the host if the path is defined with a **rcv\_method** of **PAGERECV**. This routine is defined as a TPF real-time program.

### rcv\_method

A variable specifies how messages are to be received on this path. This variable must belong to the enumeration type `s_rcv_method` defined in `claw.h`. **PAGERECV** is the only **rcv\_method** supported.

#### **PAGERECV**

CLAW calls **msg\_ep** with the address and length of the data just received. The application does not need to issue a RECEIVE to receive the data.

#### **SYNCRECV\_FLUSH**

Reserved for future IBM use.

#### **SYNCRECV\_HOLD**

Reserved for future IBM use.

#### **ASYNCRECV\_FLUSH**

Reserved for future IBM use.

#### **ASYNCRECV\_HOLD**

Reserved for future IBM use.

**AUTORECV**

Reserved for future IBM use.

**path\_anchor**

A pointer to the address of an anchor word that is unique to this particular path. This field is filled in by TPF CLAW services upon successful completion of the `claw_connect` request. The caller may use that anchor word for whatever purpose it chooses as long as the path remains connected. The address of this anchor word is passed by TPF CLAW system services as an argument in calls to the **disconn\_ep**, **msg\_ep**, and **connect\_ep** exits.

**hostappl**

A pointer to an 8-byte field that contains the host application name to be passed to the workstation on the CONNECT request.

**wsappl**

A pointer to an 8-byte field that contains the workstation application name to be passed to the workstation on the CONNECT request.

## Normal Return

Return code 0 indicates that the function was successful.

## Error Return

Following is a list of return codes that can be returned to the program that calls the `claw_connect` function. See "CLAW Return Codes" on page 405 for a complete list of the return codes and their definitions.

```
RC_CLAW_INVALID_FUNCTION
RC_CLAW_NOT_INITED
RC_CLAW_ADAPTER_NOT_OPEN
RC_CLAW_ACQUIRE_ERROR
RC_CLAW_CONNECT_ERROR
RC_CLAW_BAD_RECV_METHOD
```

## Programming Considerations

- The `claw_connect` request is blocked until the workstation issues an ACCEPT or the logical link is disconnected. An application that wants to perform a CONNECT asynchronously can create a separate task to issue the `claw_connect` function.
- If an application wants to time out a CONNECT request that was not accepted, it can create an asynchronous task to set a timer and issue a DISCONNECT request when the timer expires. The CONNECT request ends with return code `RC_CLAW_CONNECT_ERROR`. TPF CLAW services fills in the **path\_id** field immediately, before blocking.
- Activating **disconn\_ep** indicates that a DISCONNECT request is issued from the workstation to the host. No corresponding DISCONNECT request to CLAW workstation is required. The application notes that the path no longer exists. If any SEND or RECEIVE requests are active on the logical link, they are ended with a return code indicating a nonexistent path.
- Activating **msg\_ep**, which is a nonsocket message user exit (CLA4), indicates an incoming CLAW message from the workstation for the paths using the **PAGERECV recv\_method**. A new ECB is created and the parameter list is passed to the program starting at EBW000. See *TPF System Installation Support Reference* for additional information about CLA4.

## Examples

The following example issues the `claw_connect` function to open a logical link.

## claw\_connect

```
#include <claw.h>

unsigned int adapter_id;
unsigned int path_id;
char disconn_ep[5] = "CLA2";
char msg_ep[5] = "CLA4";
unsigned int pathanchor;
char host_app1[8] = "TCPIP ";
char ws_app1[8] = "API ";
int claw_rc;

/* Set up adapter_id with the value returned from the
claw_openadapter and pathid functions with the value returned from the
claw_connect function */
:
:
claw_rc = claw_connect(adapter_id,&path_id,disconn_ep,
msg_ep,PAGERECV,&pathanchor,
host_app1,ws_app1);

/*normal processing path */
:
:
```

## Related Information

- “claw\_openadapter — Initialize an Adapter” on page 397
- “claw\_accept — Accept a CONNECT Request from the Workstation” on page 386
- “claw\_disconnect — Remove a Logical Link from an Adapter” on page 393
- “claw\_send — Send a Message on an Active Logical Link” on page 403.

---

## claw\_disconnect — Remove a Logical Link from an Adapter

### ISO-C only

The `claw_disconnect` function is not available in the TARGET(TPF) C library.

The `claw_disconnect` function removes a logical link from an adapter.

### Format

```
include <claw.h>
int claw_disconnect(unsigned int adapter_id,
                   unsigned int path_id);
```

#### adapter\_id

A 4-byte field that contains the adapter ID for the adapter that was returned from the `claw_openadapter` request.

#### path\_id

A 4-byte field that contains the path ID assigned to this path when the `claw_connect` function was issued.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

Following is a list of return codes that can be returned to the program that calls the `claw_disconnect` function. See “CLAW Return Codes” on page 405 for a complete list of the return codes and their definitions.

```
RC_CLAW_INVALID_FUNCTION
RC_CLAW_NOT_INITED
RC_CLAW_ADAPTER_NOT_OPEN
RC_CLAW_ACQUIRE_ERROR
RC_CLAW_DISCONNECT_ERROR
RC_CLAW_PATH_NOT_THERE
```

### Programming Considerations

- The `claw_disconnect` function can be used to end an active logical link, to reject a pending CONNECT request from the workstation, or to end a pending CONNECT request issued from the host.
- The `claw_connect` function immediately stores the assigned **path\_id** in the **path\_id** variable supplied in the `claw_connect` function call before waiting for the workstation to accept the call. This makes it possible for another task to retrieve that value and use it in `claw_disconnect` to end the `claw_connect` function.
- After the `claw_disconnect` function is issued, the logical link is gone, as far as the application is concerned. No confirmation is received from the workstation.
- If any SEND or RECEIVE requests are active on the logical link, they end with return code RC\_CLAW\_PATH\_NOT\_THERE.

### Examples

The following example issues the `claw_disconnect` function to deactivate a logical link.

## claw\_disconnect

```
#include <claw.h>

unsigned int adapter_id;
unsigned int path_id;
int claw_rc;

/* Set up adapter_id with the value returned from the
claw_openadapter function and path_id with the value returned from
the claw_connect function */
:
    claw_rc = claw_disconnect(adapter_id,path_id);

/*normal processing path */
:
```

## Related Information

- “claw\_connect — Initiate a Request to Open a Logical Link” on page 390
- “claw\_openadapter — Initialize an Adapter” on page 397
- “claw\_accept — Accept a CONNECT Request from the Workstation” on page 386.

---

## claw\_end — Terminate All CLAW Activity

**ISO-C only**

The `claw_end` function is not available in the TARGET(TPF) C library.

The `claw_end` function ends all CLAW activity and returns all CLAW-related structures.

### Format

```
#include <claw.h>
int      claw_end(void);
```

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

Following is a list of return codes that can be returned to the program that calls the `claw_end` function. See “CLAW Return Codes” on page 405 for a complete list of the return conditions and their definitions.

```
RC_CLAW_INVALID_FUNCTION
RC_CLAW_NOT_INITED
```

### Programming Considerations

After the `claw_end` function is called, a `claw_initialization` request must be issued before any CLAW functions can be used.

### Examples

The following example issues the `claw_end` function to end all CLAW activity.

```
#include <claw.h>

int      claw_rc;
claw_rc = claw_end();
        /*normal processing path */
:
:
```

### Related Information

- “`claw_initialization` — Prepare for CLAW Activity” on page 396
- “`claw_closeadapter` — Terminate CLAW Activity on Subchannel Pair” on page 388.

---

## claw\_initialization — Prepare for CLAW Activity

**ISO-C only**

The `claw_initialization` function is not available in the TARGET(TPF) C library.

The `claw_initialization` function prepares the program for CLAW activity.

### Format

```
#include <claw.h>
int claw_initialization(const char hostname[8]);
```

**hostname**

A pointer to an 8-byte array that contains the host system name by which this host will be known to the workstation.

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

Following is a list of return codes that can be returned to the program that calls the `claw_initialization` function. See “CLAW Return Codes” on page 405 for a complete list of the return codes and their definitions.

RC\_CLAW\_INVALID\_FUNCTION  
RC\_CLAW\_INITED\_ALREADY

### Programming Considerations

- The `claw_initialization` function must be issued before any CLAW functions are requested.
- The `claw_initialization` function can be reversed only with the `claw_end` function.

### Examples

The following example issues the `claw_initialization` function to prepare for CLAW activity.

```
#include <claw.h>
int claw_rc;
char hostsys[8]= "TCPIP ";
:
:
claw_rc = claw_initialization(hostsys);
/*normal processing path */
:
:
```

### Related Information

- “`claw_openadapter` — Initialize an Adapter” on page 397
- “`claw_end` — Terminate All CLAW Activity” on page 395.

---

## claw\_openadapter — Initialize an Adapter

### ISO-C only

The `claw_openadapter` function is not available in the TARGET(TPF) C library.

The `claw_openadapter` function initializes an adapter for CLAW communications.

## Format

```
#include <claw.h>
int      claw_openadapter(unsigned int *adapter_id,
                          const char *wsname,
                          unsigned int sda_dev,
                          const char *connect_ep,
                          const char *fail_ep,
                          unsigned int *glb_anchor)
```

### adapter\_id

A pointer to a 4-byte field that contains an adapter ID with a unique 32-bit integer value that is returned by CLAW when the `claw_openadapter` function is requested. This adapter ID must be used on all subsequent CLAW requests pertaining to that subchannel pair.

### wsname

A pointer to an 8-byte field that contains the system name by that the workstation knows itself. This name is used in the system validation process.

### sda\_dev

A 2-byte field that contains the symbolic device address (SDA) on that CLAW read channel programs are processed. The SDA for the read channel is always an even number. Zero is not a valid SDA number. The SDA number for `write_dev` is equal to `sda_dev` plus one.

### connect\_ep

A pointer to a 4-byte field that contains the character string CLA3. This entry point is called asynchronously if the workstation starts a CONNECT request on this subchannel pair. This routine is defined as a TPF real-time program.

### fail\_ep

A pointer to a 4-byte field that contains the character string CLA1. This entry point is called asynchronously if the CLAW device interface detects an unrecoverable error and must shut down a subchannel pair. This call informs the caller that the adapter is no longer active. This routine is defined as a TPF real-time program.

### glb\_anchor

A pointer to the address of an anchor word unique to this particular subchannel pair that is filled in by the CLAW workstation when the `claw_openadapter` function has completed successfully. The caller may use that anchor word for whatever purpose it chooses for as long as the subchannel pair stays open. This anchor word is passed as a parameter on `connect_ep` (CLA3), `fail_ep` (CLA1), `disconn_ep` (CLA2), and `msg_ep` (CLA4) calls.

## Normal Return

Return code 0 indicates that the function was successful.

## claw\_openadapter

### Error Return

See "CLAW Return Codes" on page 405 for a complete list of the return codes and their definitions.

```
RC_CLAW_INVALID_FUNCTION
RC_CLAW_NOT_INITED
RC_CLAW_ACQUIRE_ERROR
RC_CLAW_INIT_ERROR
RC_CLAW_OPENDEVICE_ERROR
```

### Programming Considerations

- Under error conditions, calls to the **fail\_ep** (CLA1) entry points can be generated before control returns to the caller from the `claw_openadapter` function. When this happens, the caller will not have had a chance to store anything in the anchor word. To handle this condition, the `claw_openadapter` function immediately saves the value of **glb\_anchor** in the anchor word before beginning other processing. When the `claw_openadapter` function is about to return control to the calling program, it stores the address of the anchor word in **glb\_anchor**. By loading the desired contents of the anchor word into this variable before the call, the application can save itself the additional step of saving the value after the call. The application can also ensure that a correct anchor word will be passed to **fail\_ep** (CLA1) if **fail\_ep** is called during processing of the `claw_openadapter` function.
- When the program specified by **connect\_ep** (CLA3) is activated, it issues a `claw_accept` request to complete the connection. It may also use a `claw_disconnect` request to reject the connection.
- When the CLAW workstation detects an unrecoverable internal error, it shuts itself down automatically. The program specified by **fail\_ep** is activated to notify the user that a particular adapter has ended itself. This program is not activated when an adapter is shut down as a result of a `claw_closeadapter` call, or if an adapter shuts down before initialization has completed and gives an error return code from `claw_openadapter`. The application program does not have to issue a `claw_closeadapter` adapter call when this program is activated. Any connected paths on the adapter are disconnected and **disconn\_ep** (CLA2) is activated.

### Examples

The following example issues the `claw_openadapter` function to initialize an adapter.

```
#include <claw.h>

unsigned int adapter_id;
char workstation[9] = "OS2TCP ";
unsigned int sdadev;
char connectep[5] = "CLA3";
char failep[5]; = "CLA1";
unsigned int glbl_anchor;
int claw_rc;
:
:
glbl_anchor = sdadev; /* setup glb_anchor value */
claw_rc = claw_openadapter(&adapter_id,workstation,
                          sdadev,connectep,failep,&glbl_anchor);
```

⋮                   /\*normal processing path \*/

## **Related Information**

- “claw\_closeadapter — Terminate CLAW Activity on Subchannel Pair” on page 388.

## claw\_query — Get the Status of CLAW Adapter or Logical Links

**ISO-C only**

The claw\_query function is not available in the TARGET(TPF) C library.

The claw\_query function gets information about the status of active CLAW adapters or logical links.

### Format

```
#include <claw.h>
int claw_query(unsigned int adapter_id,
               enum query_level query_lvl,
               union query_return *buffer_addr);
```

**adapter\_id**

A 4-byte field that contains the adapter ID for the adapter that was returned from the claw\_openadapter request.

**query\_lvl**

One of the following values that describes the type of QUERY:

**QTYPE\_LEVEL1**

Returns the adapter ID of the next active adapter control block on the active queue. See Table 10 for more information about QTYPE\_LEVEL1.

**QTYPE\_LEVEL2**

Obtains information about active CLAW adapters. See Table 11 for more information about QTYPE\_LEVEL2.

**QTYPE\_LEVEL3**

Obtains information about active logical path 1 on an open CLAW adapter. See Table 12 on page 401 for more information about QTYPE\_LEVEL3.

**buffer\_addr**

The pointer to a structure of type query\_return, which is defined in claw.h where the query response will be placed. The size and format of this buffer depends on the query\_lvl value.

Following is the buffer returned for query level 1:

Table 10. CLAW Query Level 1 Buffer

Offset	Length	Name	Description
0	4	CLAW_Q1_ADAPID	Returned adapter ID

Following is the buffer returned for query level 2:

Table 11. CLAW Query Level 2 Buffer

Offset	Length	Name	Description
0	8	CLAW_Q2_HOSTNAME	Host system name
8	8	CLAW_Q2_WSNAME	Workstation system name
10	4	CLAW_Q2_MSENT	Total messages sent
14	4	CLAW_Q2_MRECV	Total messages received
18	4	CLAW_Q2_BSENTHI	Total bytes sent (high-order fullword)

Table 11. CLAW Query Level 2 Buffer (continued)

Offset	Length	Name	Description
1C	4	CLAW_Q2_BSENTLO	Total bytes sent (low-order fullword)
20	4	CLAW_Q2_BRECVHI	Total bytes received (high-order fullword)
24	4	CLAW_Q2_BRECVLO	Total bytes received (low-order fullword)
28	4	CLAW_Q2_TIME	Time of last STAT
2C	4	CLAW_Q2_PATHS	Number of open paths
30	4	CLAW_Q2_STATUS	Status bits (not used)
34	4	CLAW_Q2_READPAGES	Pages allocated for reads
38	4	CLAW_Q2_WRITEPAGES	Pages allocated for writes
3C	4	CLAW_Q2_READDEV	SDA (Must be an even number)
40	4	CLAW_Q2_RESERVE	IBM Reserved

Following is the buffer returned for query level 3:

Table 12. CLAW Query Level 3 Buffer

Offset	Length	Name	Description
0	8	CLAW_Q3_HOSTAPPL	Host application
8	8	CLAW_Q3_WSAPPL	Workstation application
10	4	CLAW_Q3_MSENT	Total messages sent
14	4	CLAW_Q3_MRECV	Total messages received
18	4	CLAW_Q3_BSENTHI	Total bytes sent (high-order fullword)
1C	4	CLAW_Q3_BSENTLO	Total bytes sent (low-order fullword)
20	4	CLAW_Q3_BRECVHI	Total bytes received (high-order fullword)
24	4	CLAW_Q3_BRECVLO	Total bytes received (low-order fullword)
28	4	CLAW_Q3_TIME	Time of last STAT
2C	4	CLAW_Q3_PATHID	Path ID
30	1	CLAW_Q3_STATUS	Status bits
		CLAW_Q3_CLPTSFRE	X'00'—Path is free
		CLAW_Q3_CLPTSHO	X'01'—Host connect pending
		CLAW_Q3_CLPTSWCO	X'02'—Workstation connect pending
		CLAW_Q3_CLTSACT	X'03'—Path is connected pending
31	3	CLAW_Q3_MISC	IBM reserved
34	4	CLAW_Q3_PENDSENDS	Active send request
38	4	CLAW_Q3_PENDRECVS	Receive request active

## claw\_query

### Normal Return

Return code 0 indicates that the function was successful.

### Error Return

Following is a list of return codes that can be returned to the program that calls the `claw_query` function. See “CLAW Return Codes” on page 405 for a complete list of the return codes.

```
RC_CLAW_INVALID_FUNCTION
RC_CLAW_NOT_INITED
RC_CLAW_ADAPTER_NOT_OPEN
RC_CLAW_BAD_QUERY_TYPE
```

### Programming Considerations

Following are the programming considerations for CLAW query levels 1 and 3. There are no programming considerations for CLAW query level 2.

#### QTYPE\_LEVEL1

- If the supplied adapter ID parameter is zero, the adapter ID of the first open adapter is returned in `CLAW_Q1_ADAPID`. If there is no open adapter, zero is returned in `CLAW_Q1_ADAPID`.
- If the supplied adapter ID parameter is the adapter ID of the valid open adapter, the adapter ID of the next open adapter in the list is returned. If this is the last adapter in the list, zero is returned.
- If the supplied adapter ID parameter is not a valid adapter ID for any adapter in the list, -1 is returned.

#### QTYPE\_LEVEL3

If logical path 1 is not active for the corresponding adapter ID (that is, `CLPTSFRE` in `CLAW_Q3_STATUS` is on), no other fields in the structure are valid.

### Examples

The following example returns the information about active CLAW adapters.

```
include <claw.h>

unsigned int adapter_id;
union query_return buffer;
int claw_rc;

/* Set up adapter_id with the value returned from the
claw_openadapter function */

claw_rc = claw_query(adapter_id,QTYPE_LEVEL2,&buffer);

/*normal processing path */
:
```

### Related Information

None.

## claw\_send — Send a Message on an Active Logical Link

### ISO-C only

The `claw_send` function is not available in the TARGET(TPF) C library.

The `claw_send` function sends a message on an active logical link.

### Format

```
#include <claw.h>
int claw_send(unsigned int adapter_id,
              unsigned int path_id,
              char *msg_addr,
              int msg_len,
              enum s_send_method send_method);
```

#### adapter\_id

A 4-byte field that contains the adapter ID for this adapter that was returned by the `claw_openadapter` request.

#### path\_id

A 4-byte field that contains the path ID assigned to this path at `claw_connect` time.

#### msg\_addr

A pointer to a data buffer or a list of discontinuous buffers. If **send\_method** for this message is **SEND\_LIST**, this field contains the address of the first element in an array of CLAWPAGE structures. This structure describes the address, length, and more-to-come bit status of a page frame to be transferred. If **send\_method** for this message is **SEND\_NOLIST**, this field contains the address of the message to be sent.

#### msg\_len

A 4-byte field that contains the number of bytes in a data buffer or the number of elements in an array of pointers. If **send\_method** for this message is **SEND\_LIST**, this field contains the number of elements in an array of CLAWPAGE structures. If **send\_method** for this message is **SEND\_NOLIST**, this field contains the length of the message to be sent up to 4 KB.

#### send\_method

A variable specifying how messages are to be sent on this path. This variable must belong to the enumeration type `s_send_method`, defined in the `claw.h` header file. The following values may be specified:

##### SEND\_LIST

The application passes an array of descriptors to TPF CLAW services, each containing the address, length, and status of a message or portion of a message. A message or portion of a message described by one of these descriptors can be any length but it must not cross a 4 KB boundary (thereby effectively limiting its length to 4096 bytes).

##### SEND\_NOLIST

The application passes the address and length of the message to CLAW. The message must not cross a 4 KB boundary.

### Normal Return

Return code 0 indicates that the function was successful.

## claw\_send

### Error Return

Following is a list of return codes that can be returned to the program that calls the `claw_send` function. See “CLAW Return Codes” on page 405 for a complete list of the return codes.

```
RC_CLAW_INVALID_FUNCTION
RC_CLAW_NOT_INITED
RC_CLAW_ADAPTER_NOT_OPEN
RC_CLAW_ACQUIRE_ERROR
RC_CLAW_PATH_NOT_THERE
RC_CLAW_BAD_SEND_METHOD
```

### Programming Considerations

- The `claw_send` request is blocked until all requested data has been transferred. An active `claw_send` request can be ended by the `claw_disconnect` or `claw_closeadapter` functions, but some or all of the message may be sent anyway. If **SEND\_LIST** is used for **send\_method**, a logical message consists of a series of possible discontinuous blocks, as specified in the list, until one without the more-to-come bit set on is found. It is possible to have one logical message span several `claw_send` requests. It is also possible to have one `claw_send` request include multiple messages.
- The CLAWPAGE structure ICLAWG DSECT is used to communicate between the CLAW application and CLAW when the **send\_method** is **SEND\_LIST**. See the ICLAWG DSECT for more information.

### Examples

The following example issues the `claw_send` function to send the message to the workstation.

```
#include <claw.h>

    unsigned int adapter_id;
    unsigned int path_id;
    char        *msg;
    int         length;
    int         claw_rc;

/* Set up adapter_id with the value returned from the claw_openadapter function
and path_id with the value returned from the claw_connect function */
:
:
claw_rc = claw_send(adapter_id,path_id,msg,length,SEND_NOLIST);
           /*normal processing path */
:
:
```

### Related Information

- “`claw_closeadapter` — Terminate CLAW Activity on Subchannel Pair” on page 388
- “`claw_connect` — Initiate a Request to Open a Logical Link” on page 390
- “`claw_disconnect` — Remove a Logical Link from an Adapter” on page 393
- “`claw_openadapter` — Initialize an Adapter” on page 397.

## CLAW Return Codes

The following table shows the return codes defined for calls to the CLAW workstation.

Table 13. CLAW Return Codes Defined for CLAW Functions

Symbolic Name	Hex Value	Description
RC_CLAW_GOOD_RETURN	0000	The return is correct.
RC_CLAW_INVALID_FUNCTION	0001	CLAW was called with a first argument that is not valid. This argument must be one of the CLAW restricted functions.
RC_CLAW_INIT_ALREADY	0006	A <code>claw_initialization</code> function was issued, but a previous <code>claw_initialization</code> had already been issued.
RC_CLAW_NOT_INITED	0007	A function was issued before the <code>claw_initialization</code> function.
RC_CLAW_ADAPTER_NOT_OPEN	000A	The caller must specify an adapter ID parameter on several CLAW calls. This must be the value that was returned on a <code>claw_openadapter</code> call. If an incorrect value is specified, TPF CLAW services returns this error code.
RC_CLAW_ACQUIRE_ERROR	000B	During the processing of a CLAW function call to a CLAW workstation, TPF CLAW system services was unable to obtain a control block needed to process the function call.
RC_CLAW_CONNECT_ERROR	000D	An internal error occurred during processing of a <code>claw_connect</code> request.
RC_CLAW_DISCONNECT_ERROR	000E	An internal error occurred during processing of a <code>claw_disconnect</code> request.
RC_CLAW_PATH_NOT_THERE	000F	An incorrect path ID was specified for a CLAW request for that this parameter is required. Valid path IDs are passed to the user program in the <b>connect_ep</b> exit or returned to it after a <code>claw_connect</code> call, and remain valid until the path is ended using a DISCONNECT request. This return code may also be generated for a <code>claw_connect</code> request if the host or the workstation ends the pending path with a DISCONNECT request.
RC_CLAW_BAD_SEND_METHOD	0010	A <code>claw_send</code> request was issued with a <b>send_method</b> that was not valid.
RC_CLAW_BAD_RECV_METHOD	0011	A <code>claw_connect</code> or <code>claw_accept</code> request was issued with a <b>recv_method</b> value that was not valid.
RC_CLAW_SEND_ERROR	0019	A <code>claw_send</code> request encountered an error condition from the CLAW device interface.
RC_CLAW_INIT_ERROR	001B	This error is returned from a <code>claw_openadapter</code> request. It indicates that the CLAW system validate processing did not complete successfully. A CLAW error log is generated, providing more information about the error.
RC_CLAW_BAD_QUERY_TYPE	001C	This error is returned from a <code>claw_query</code> request. It indicates that the <b>query_lvl</b> parameter in the request was not valid.
RC_CLAW_DUPLICATE_ADAPTER	001D	The call specified an adapter ID on the <code>claw_openadapter</code> function and the adapter is already active.
RC_CLAW_IO_OUTSTANDING	001E	A <code>claw_send</code> request was issued and there is an I/O outstanding on the ECB.

## claw\_send

Table 13. CLAW Return Codes Defined for CLAW Functions (continued)

Symbolic Name	Hex Value	Description
RC_CLAW_FAIL_DURING_OPEN	001F	An error occurred in the CLAW device interface that caused the adapter to close.
RC_CLAW_OPENDEVICE_ERROR	1000	TPF CLAW services uses the MSDAC macro call to mount the Read and Write devices specified in the <code>claw_openadapter</code> call. If MSDAC returns an error return code, this error code is returned as the return code from <code>claw_openadapter</code> .

---

## Appendix F. Using the Internet Protocol Trace Facility

The Internet Protocol (IP) trace facility provides a detailed trace of the data transferred between the TPF system and remote resources connected through IP networks. Each time the TPF system sends or receives data, some, all, or none of that data is stored in an entry in the IP trace table depending on the resources that you are tracing. Later, you can display the entries in the IP trace table online or write the IP trace table to a real-time tape and create a report by using the IPTPRT program to view or print offline.

The data stored in an IP trace table entry includes the following parts of the IP packet:

- The entire IP header
- The entire protocol (Transmission Control Protocol (TCP) or User Datagram Protocol (UDP)) header, if one exists
- A user-defined amount of the user data in the packet.

You can trace the data for all of the resources in the network or for only specific resources in the network. In addition, you can set up an individual IP trace. See "Using the Individual IP Trace Function" on page 76.

---

### About the IP Trace Table

The IP trace table resides in core storage in the TPF system. The number of entries in the trace table varies depending on how many bytes of user data from each IP packet are being stored in the trace table.

The IP trace table operates in wraparound mode; that is, once all of the entries in the IP trace table are used, the TPF system begins to overwrite the oldest entries with the new data entries.

To store data in the IP trace table, you must first start the IP trace facility and specify which resources you want to trace. When you no longer want to trace data, you can stop the IP trace facility.

---

### Starting the IP Trace Facility and Specifying Which Data to Trace

Use the ZTTCP TRACE command to start the IP trace facility and specify which resources to trace. The TPF system stores the data being traced in entries in the IP trace table.

You can use the IP trace facility to trace the following:

- Routing Information Protocol (RIP) messages, or you can select to not trace them
- All the data transferred between the TPF system and remote resources
- Only data for a specific channel data link control (CDLC) IP router
- Only data for a specific OSA-Express connection
- Only data for sockets associated with a specific local IP address.

You can start more than one trace at the same time; for example, you can trace the data for IP router X and local IP address 1.1.1.1. Depending on how you define your traces, some traces may overlap other traces. For example, if you start tracing

the data for IP router X and then start tracing the data for the local IP address later, the trace defined for the local IP address will overlap the trace defined for IP router X. In addition, if you have traces defined for specific resources and then start tracing all resources later, the trace defined for all resources will overlap the traces defined for specific resources.

At any time, you can display information about the status of the IP trace facility and the data being traced. See “Displaying Information about the IP Trace Facility” on page 409 for more information.

To start the IP trace facility, do the following:

1. Enter one or more ZTTCP TRACE commands with the START parameter specified to start the IP trace facility and specify which resources you want to trace.
2. Enter the ZTTCP TRACE command with the SIZE parameter specified to specify how many bytes of user data in each IP packet to store in the IP trace table. See “Defining How Much Data to Store in the IP Trace Table” for more information.
3. Enter **ZTTCP TRACE START TAPE** to start writing the IP trace table to a real-time tape. See “Writing the IP Trace Table to a Real-Time Tape” on page 409 for more information.

---

## Stopping the IP Trace Facility

Use the ZTTCP TRACE command with the STOP parameter specified to stop the IP trace facility. You can stop the IP trace facility for particular traces that you defined. See “Starting the IP Trace Facility and Specifying Which Data to Trace” on page 407 for more information about defining traces.

To stop tracing all resources in the TPF system, but to continue the traces that you defined for specific resources, enter **ZTTCP TRACE STOP ALL**. For example, assume you were tracing data for IP router X and then started tracing all resources in the TPF system. After you enter **ZTTCP TRACE STOP ALL**, the IP trace facility continues to trace data for only IP router X.

To stop a trace that was defined for a specific resource, enter the ZTTCP TRACE command with the STOP parameter and the IP or SDA parameter specified. If the trace that you stopped overlapped with another trace, the IP trace facility will continue that other trace. For example, assume you were tracing IP router X and then started tracing the local IP address associated with IP router X. If you stop the trace for that local IP address, the IP trace facility will continue tracing IP router X.

---

## Defining How Much Data to Store in the IP Trace Table

For each IP packet that is traced, the IP trace facility saves the following in the IP trace table entry:

- The entire IP header
- The entire protocol (TCP or UDP) header, if one exists
- A user-defined amount of the user data in the packet.

Use the ZTTCP TRACE command with the SIZE parameter specified to increase or decrease how much user data in each IP packet is traced. If an IP packet is traced and the amount of user data in the packet is more than the maximum size specified on the ZTTCP TRACE command, only the first *n* bytes of user data are placed in

the IP trace table (where  $n$  is the size specified for the ZTTCP TRACE command). See *TPF Operations* for more information about the ZTTCP TRACE command.

---

## Writing the IP Trace Table to a Real-Time Tape

Before you can create an IPTPRT report to view or print offline, you must write the IP trace table to a real-time tape. See “Using the Offline IPTPRT Utility to Create an IPTPRT Report” on page 410 for more information about creating an IPTPRT report.

To write the IP trace table to a real-time tape, do the following:

1. Ensure you have a real-time tape mounted for the TPF system. See *TPF Operations* for more information about mounting a tape.
2. Enter one or more ZTTCP TRACE commands with the START parameter specified to start the IP trace facility and specify which data you want to trace. See “Starting the IP Trace Facility and Specifying Which Data to Trace” on page 407 for more information.
3. Enter **ZTTCP TRACE START TAPE** to start writing the IP trace table to the real-time tape.

The IP trace facility will automatically write each 4-KB block of the IP trace table to the real-time tape when that 4-KB block becomes full. If the length of the queue to the real-time tape becomes too large, the IP trace facility stops writing the IP trace table to tape until the queue becomes smaller.

When you are ready to create an IPTPRT report, remove the real-time tape so that it can be processed by the IPTPRT utility. See *TPF Operations* for more information about the ZTTCP TRACE command or removing a real-time tape.

---

## Displaying Information about the IP Trace Facility

Use the ZTTCP DISPLAY command to display information about the IP trace facility.

Enter **ZTTCP DISPLAY ALL** to display the status of each IP router. The TRACE column of the output display shows whether IP trace is active for a given IP router.

Enter **ZTTCP DISPLAY LOCIPS** to display the status of each local IP address. The TRACE column of the output display shows whether IP trace is active for a given local IP address.

See *TPF Operations* for more information about the ZTTCP DISPLAY command and for an example of the informational display.

---

## Displaying the IP Trace Table Online

Use the ZIPTR command to display the IP trace table online. You can display all of the trace table or only part of the trace table. You can also display formatted entries in the IP trace table.

Enter the ZIPTR command with the ALL parameter specified to display all of the IP trace table entries. When you display all of the trace table entries, the oldest entry (that is, the entry with the oldest time stamp) is displayed first, followed by the next oldest entry, and so on. Therefore, the last entry displayed is the newest entry in the IP trace table.

You can also enter the ZIPTR command with the num parameter specified to specify the number of IP trace table entries that you want to display. When you do this, the newest entries in the trace table are displayed.

Remember that the IP trace facility can continue to store new data in the IP trace table while you are displaying entries from the trace table. Therefore, the newest entries continue to change as more data is stored in the trace table. For example, if you display 11 entries from the IP trace table, wait a few seconds while data continues to flow, and then you display 11 entries from the IP trace table again, the 11 entries displayed the second time are different from the 11 entries displayed the first time.

When you enter the ZIPTR command to display the IP trace table, the last line of the information displayed tells you how many entries are in the trace table.

There are two formats in which you can display the IP trace table. You can create a compacted display of the trace table or a formatted display of the trace table.

## Creating a Compacted Display of the IP Trace Table

To create a compacted display of the IP trace table, enter the ZIPTR command without the FORMAT parameter specified.

In the compacted display of the IP trace table, each trace table entry is displayed on a separate line and only *part* of the user data being traced is included.

## Creating a Formatted Display of the IP Trace Table

To create a formatted display of the IP trace table, enter the ZIPTR command with the FORMAT parameter specified.

In the formatted display of the IP trace table, each trace table entry is formatted and *all* the user data being traced is displayed.

Different information is displayed depending on the protocol of the packet (TCP, UDP, or others).

See *TPF Operations* for more information about the ZIPTR command and for an example of the informational display.

---

## Using the Offline IPTPRT Utility to Create an IPTPRT Report

Use the offline IPTPRT utility to create an IPTPRT report, which you can view or print offline. Unlike using the ZIPTR command to display the IP trace table online, the IPTPRT utility offers you more flexibility in selecting the information to include in the IPTPRT report. For example, you can print in the IPTPRT report only the data transferred between the TPF system and a specific IP router, or only the data between the TPF system and a specific remote resource. See "Defining the IPTPRT Report" on page 411 for more information.

Another difference between creating an IPTPRT report and using the ZIPTR command to display the IP trace table online is that you create an IPTPRT report from the IP trace table on a real-time tape rather than in core storage. If you enter the ZIPTR command to display the IP trace table online while you are tracing active resources, the oldest entries in the IP trace table may be overwritten with new data.

Therefore, after this happens, you cannot display those entries online. However, because you create an IPTPRT report from the IP trace table on a real-time tape, you never have this problem.

The IPTPRT utility runs on an MVS system. Before you can use the IPTPRT utility to create an IPTPRT report, you must do the following:

1. Compile the IPTPRT utility.
2. Submit the object code to the object library.
3. Link the object code to the link library.

To create an IPTPRT report, do the following:

1. Follow the steps in “Starting the IP Trace Facility and Specifying Which Data to Trace” on page 407 to start the IP trace facility and specify which data you want to trace.
2. When you are ready to create an IPTPRT report, perform a tape switch for the real-time tape. See *TPF Operations* for more information about performing a tape switch.
3. Create the job control language (JCL) needed to run the IPTPRT utility. See “Sample JCL for the IPTPRT Utility” for an example.
4. Define the IPTPRT report by updating the PARM= parameter in the IPTPRT JCL. This allows you to specify the format of the IPTPRT report and the data you want to include in it. See “Defining the IPTPRT Report” for more information.
5. Submit the IPTPRT JCL to the MVS system to run the IPTPRT utility and create the IPTPRT report. See “IPTPRT Messages” on page 421 for information on possible return codes.
6. View or print the IPTPRT report.

## Sample JCL for the IPTPRT Utility

Figure 36 shows an example of the JCL that you can use to run the IPTPRT utility. Change the tape number, shown as XXXXXX, to the tape number for the real-time tape that contains the IP trace table. Change the link library name, shown as NNN.NNNN.NNNN.NN, to the name of your link library.

```
//IP EXEC PGM=IPTPRT,PARM='ALL COMPACT'  
//STEPLIB DD DISP=SHR,DSN=NNN.NNNN.NNNN.NN  
//PRINT DD SYSOUT=A,DCB=(LRECL=133,BLKSIZE=3990,RECFM=FBA)  
//IPTR DD DSN=RTL,DCB=(LRECL=4095,BLKSIZE=32760,RECFM=U),  
// DISP=OLD,LABEL=(2,BLP),UNIT=TAPE,VOL=SER=XXXXXX  
//SYSUDUMP DD SYSOUT=A  
/*  
/* RECFM=VB FOR TAPES CREATED IN BLOCKED FORMAT.
```

Figure 36. JCL for the IPTPRT Utility

## Defining the IPTPRT Report

Use the PARM= parameter in the IPTPRT JCL to define the IPTPRT report. You can specify the data that you want to print in the IPTPRT report as well as how you want to format the IPTPRT report.

Unlike using the ZIPTR command to display a specific number of entries from the IP trace table, you can actually define the type of data that you want to print in an IPTPRT report. For example, you can print the entire IP trace table on the real-time

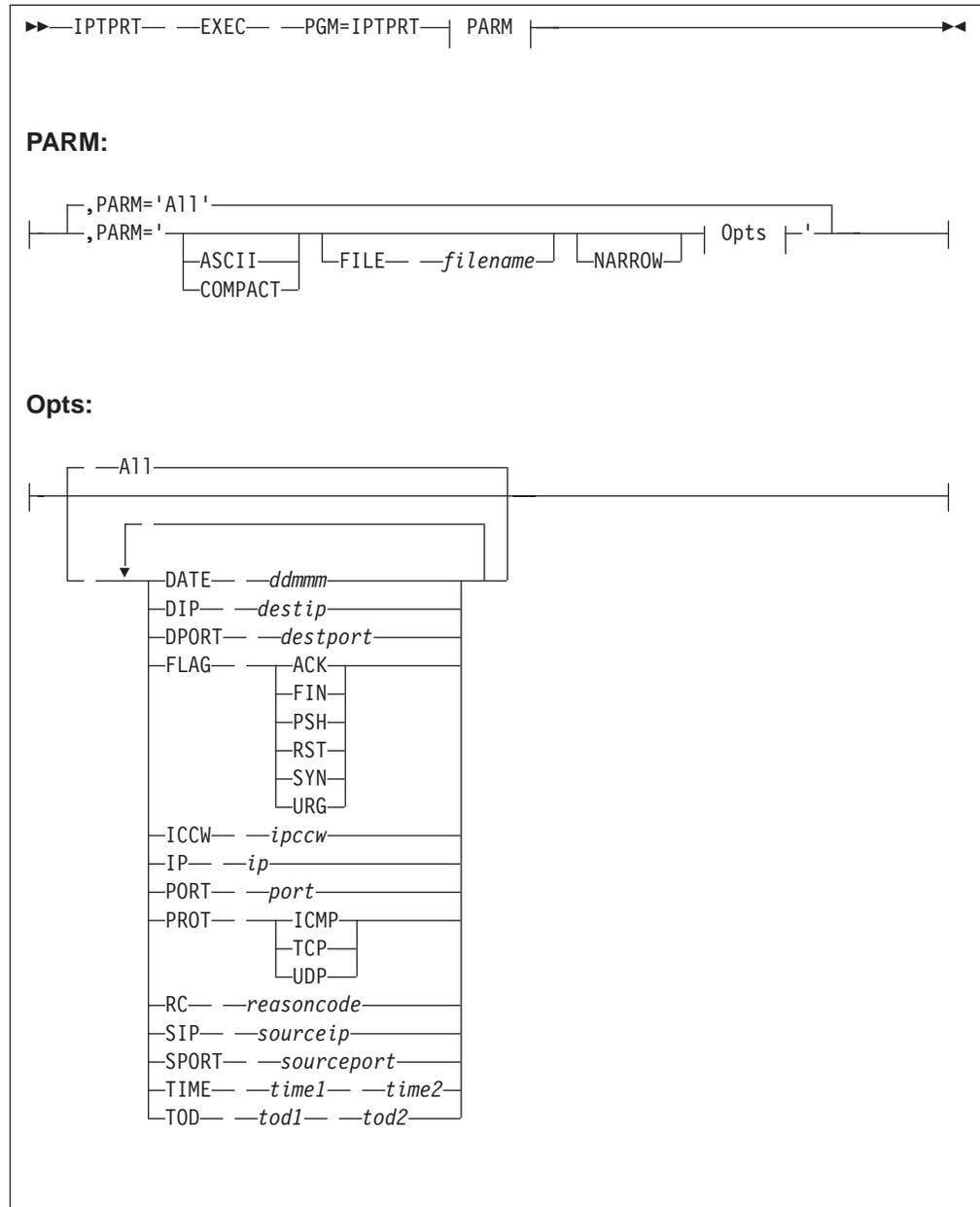
tape or you can print only the data that flowed over an IP router. You can also print only the data for a remote resource or print only the data that has a time stamp in a specified range.

There are two formats in which you can create the IPTPRT report. Use the PARM= parameter to specify whether you want to create a compacted IPTPRT report or a formatted IPTPRT report.

See “Sample Compacted IPTPRT Report” on page 416 for an example of a compacted IPTPRT report and “Sample Formatted IPTPRT Report” on page 418 for an example of a formatted IPTPRT report.

### **PARM= Parameter for the IPTPRT JCL**

Many values are available in the IPTPRT JCL for the PARM= parameter that allow you to change the contents of the IPTPRT report to your specific needs. The following shows the syntax for the PARM= parameter and describes the values.



**ALL**

includes all of the IP packets in the IP trace table in the IPTPRT report.

**ASCII**

displays the data portion of the output in ASCII format. This value applies only to a formatted IPTPRT report.

**COMPACT**

creates a compacted IPTPRT report in which each entry is printed on a single line. See “Sample Compacted IPTPRT Report” on page 416 for an example of a compacted IPTPRT report. If you do not specify this value, a formatted IPTPRT report is created.

**DATE dmmm**

includes in the IPTPRT report only the IP packets that flowed on the specified date, where *dd* is the day and *mmm* is the first 3 characters of the name of the month.

**DIP** *destip*  
includes in the IPTPRT report only the IP packets whose destination IP address is *destip*.

**DPORT** *destport*  
includes in the IPTPRT report only the IP packets whose destination port is *destport*.

**FILE** *filename*  
writes the IPTPRT report to the specified file.

**FLAG** *flagname*  
includes in the IPTPRT report only the IP packets for TCP sockets that have the specified flags set in the TCP header of the packet.

**ICCW** *ipccw*  
includes in the IPTPRT report only the IP packets that flowed between the TPF system and the IP router or OSA-Express connection whose IP CCW index is *ipccw*.

**IP** *ip*  
includes in the IPTPRT report only the IP packets whose destination IP address is *ip*, or whose source IP is *ip*.

**NARROW**  
creates the IPTPRT report in a narrow format (80 columns wide). If you do not specify this value, the default is to create an IPTPRT report that is 132 columns wide.

**PORT** *port*  
includes in the IPTPRT report only the IP packets whose destination port is *port*, or whose source PORT is *port*.

**PROT** *protocol*  
includes in the IPTPRT report only the IP packets whose protocol is *protocol*.

**RC** *reasoncode*  
includes in the IPTPRT report only the IP packets that contain a predefined reason code that indicates an exception condition is associated with the packet, where *reasoncode* is one of the following:

- 00** includes all possible reason codes.
- 01** includes all packets with the REJECTED BY FIREWALL reason code value. This reason code occurs when a packet is rejected based on a packet filtering rule. This reason code is shown in the IPTPRT report for both the input packet that generated the exception condition and the output packet that is sent as a result of the exception condition.
- 02** includes all packets with the DISCARDED BY FIREWALL reason code value. This reason code occurs when a packet is discarded based on a packet filtering rule.
- 03** includes all packets with the SERVER NOT ACTIVE reason code value. This reason code occurs when a TCP connection request is received for a server that is not active. This reason code is shown in the IPTPRT report for both the input packet that generated the exception condition and the output packet that is sent as a result of the exception condition.
- 04** includes all packets with the SOCKET DOES NOT EXIST reason code value. This reason code occurs when a TCP message (not a connection request) was received, but the specified socket does not exist. This reason

code is shown in the IPTPRT report for both the input packet that generated the exception condition and the output packet that is sent as a result of the exception condition.

- 05** includes all packets with the BACKLOG LIMIT EXCEEDED reason code value. This reason code occurs when the remote client tries to start a connection with a TCP server on the TPF system, but the backlog limit for this application has been exceeded. This reason code is shown in the IPTPRT report for both the input packet that generated the exception condition and the output packet that is sent as a result of the exception condition.
- 06** includes all packets with the NO SOCKETS AVAILABLE reason code value. This reason code occurs when a TCP connection request was received, but no socket block entries are available in the TPF system to start a new socket. This reason code is shown in the IPTPRT report for both the input packet that generated the exception condition and the output packet that is sent as a result of the exception condition.
- 07** includes all packets with the POSSIBLE SYN ATTACK reason code value. This reason code occurs when the TPF system is running out of socket blocks and the connection request from this remote client has been pending for a long period of time. The connection request is cleaned up to free socket block entries.
- 08** includes all packets with the CLOSED BY APPLICATION reason code value. This reason code occurs when the `close` function was issued for this socket. The socket was either starting, ending, or had input messages queued that have not been processed.
- 09** includes all packets with the CLOSED BY SOCKET SWEEPER reason code value. This reason code occurs when the socket sweeper program closes a socket because the socket is no longer being used.
- 10** includes all packets with the RETRANSMIT LIMIT EXCEEDED reason code value. This reason code occurs when the TPF system closes the socket because the TPF system retransmitted messages that have not been acknowledged by the remote client and the retransmit limit has been reached.
- 11** includes all packets with the ZTTCP INACTIVATE SOCKETS reason code value. This reason code occurs when the ZTTCP INACTIVATE command is entered with the SOCKETS parameter specified to deactivate the socket.
- 12** includes all packets with the ZSOCK INACTIVATE SOCKETS reason code value. This reason code occurs when the ZSOCK command was entered with the INACT parameter specified to deactivate one or more sockets.
- 13** includes all packets with the NOT AUTHORIZED reason code value. This reason code occurs when the TCP/IP native stack support accept connection user exit, UACC, rejected the connection request.
- 14** includes all packets with the CYCLE DOWN reason code value. This reason code occurs when a socket is closed because the TPF system is cycling down to 1052 state.
- 15** includes all packets with the TCP OPTIONS NOT VALID reason code value. This reason code occurs when the connection request was received, but was rejected because the TCP options specified by the remote client are not valid. This reason code is shown in the IPTPRT report for both the

input packet that generated the exception condition and the output packet that is sent as a result of the exception condition.

- 16** includes all packets with the SSL DAEMON SHUTDOWN reason code value. This reason code occurs when the socket is associated with a shared SSL session and the SSL daemon processes are shutting down and closing their sockets.
- 17** includes all packets with the APPLICATION NOT ACTIVE reason code value. This reason code occurs when a UDP message was received, but the specified application (port) is not active. This reason code is shown in the IPTPRT report for both the input packet that generated the exception condition and the output packet that is sent as a result of the exception condition.
- 18** includes all packets with the DESTINATION NOT TPF reason code value. This reason code occurs when a packet is received from the network, but the destination is not an IP address in the TPF system and the message is discarded.
- 19** includes all packets with the RESTRICTED CDLC IP ADDRESS reason code value. This reason code occurs when the packet is discarded by the TPF system because the packet was received on a restricted IP address, but across the wrong symbolic device address (SDA).
- 20** includes all packets with the RETRANSMITTED MESSAGE reason code value. This reason code occurs when the message is retransmitted by the TPF system.

**SIP** *sourceip*

includes in the IPTPRT report only the IP packets whose source IP address is *sourceip*.

**SPORT** *sourceport*

includes in the IPTPRT report only the IP packets whose source port is *sourceport*.

**TIME** *time1 time2*

includes in the IPTPRT report only the IP packets in the specified time-stamp range, where *time1* and *time2* are the beginning and ending times in the format *hh.mm.ss*.

**TOD** *tod1 tod2*

includes in the IPTPRT report only the IP packets in the specified time-stamp range, where *tod1* and *tod2* are the beginning and ending times in time-of-day clock format.

### Sample PARM= Parameters for the IPTPRT JCL

A compacted IPTPRT report of all the data in the IP trace table is created in the following example.

```
//IP EXEC PGM=IPTPRT,PARM='ALL COMPACT'
```

A formatted IPTPRT report is created in the following example. The report includes all data transferred between the TPF system and the remote resource whose IP address is 9.117.102.113.

```
//IP EXEC PGM=IPTPRT,PARM='IP 9.117.102.113'
```

### Sample Compacted IPTPRT Report

To create a compacted IPTPRT report, specify the COMPACT value in the PARM= parameter of the IPTPRT JCL.

In a compacted IPTPRT report, each IP packet is printed on a single line and only part of the user data that was traced is included. The following information is displayed:

RW	The read/write operation code, where: EVEN            Even numbers represent read operations. ODD             Odd numbers represent write operations.
IN	The IP channel command word (IPCCW) area index, where: 01–C7           CDLC IP routers. D1–EE           OSA-Express connections. FF               Local sockets.
SOURCE IP	The source IP address.
DEST IP	The destination IP address.
SPORT	The source port. This field has meaning only for packets using TCP or UDP.
DPORT	The destination port. This field has meaning only for packets using TCP or UDP.
PR	The protocol. Sample values are as follows: 01               ICMP 06               TCP 11               UDP
FG	The TCP flag byte. This field has meaning only for packets using TCP.
DATA	The user data in the IP packet.

Figure 37 on page 418 shows a narrow format example of a compacted IPTPRT report.

```

*****
TRANSACTION PROCESSING FACILITY TCP/IP TRACE OUTPUT
*****
RECORDS MATCHING THE FOLLOWING SELECTION CRITERIA WILL BE PRINTED:
PROTOCOLS: . . . . . ALL
SOURCE PORTS: . . . . . ALL
DESTINATION PORTS: . . . . ALL
SOURCE IP ADDRESSES: . . . ALL
DESTINATION IP ADDRESSES: ALL
REASON CODES: . . . . . ALL
IP CCW: . . . . . ALL
DATE: . . . . . FROM JAN01 TO DEC31
TIME: . . . . . FROM 00:00:00 TO 23:59:59
TOD (FIRST WORD): . . . . FROM 00000000 TO FFFFFFFF
TCP FLAGS: . . . . . ALL
NARROW LAYOUT
COMPACT FORMAT
RW IN  SOURCE IP      DEST IP      SPORT DPORT PR FG DATA
32 03  9.117.249.50    9.117.249.51  1024  9999 06 02
31 03  9.117.249.51    9.117.249.50  9999  1024 06 12
52 03  9.117.249.50    9.117.249.51  1024  9999 06 10
32 03  9.117.249.50    9.117.249.51  1024  9999 06 18 D7C9D5C760D7D6D5
32 03  9.117.249.50    9.117.249.51  1024  9999 06 18 4040F1F0F0818181
51 03  9.117.249.51    9.117.249.50  9999  1024 06 10
31 03  9.117.249.51    9.117.249.50  9999  1024 06 18 4040F1F0F0828282
52 03  9.117.249.50    9.117.249.51  1024  9999 06 18 4040F1F0F0818181

```

Figure 37. Compacted IPTPRT Report

### Sample Formatted IPTPRT Report

To create a formatted IPTPRT report, do not specify the COMPACT value in the PARM= parameter of the IPTPRT JCL.

In a formatted IPTPRT report, each IP packet is formatted and all the user data that was traced is printed.

Different information is displayed depending on the protocol of the packet (TCP, UDP, or others).

- For IP packets using TCP, the following information is displayed:

**RWI**

The read/write operation code, where:

**EVEN** Even numbers represent read operations.

**ODD** Odd numbers represent write operations.

**IPCCW**

The IP channel command word (IPCCW) area index, where:

**01–C7** CDLC IP routers.

**D1–EE**

OSA-Express connections.

**FF** Local sockets.

**SOURCE IP**

The source IP address.

**DEST IP**

The destination IP address.

LEN  
The length of the IP packet.

TOD  
The time stamp.

PROTOCOL  
The protocol of the IP packet, which for TCP is X'06'.

SOURCE PORT  
The source port.

DEST PORT  
The destination port.

SEQ  
The sequence number in the TCP header.

ACK  
The acknowledgment number in the TCP header.

WINDOW  
The window size in the TCP header.

URGENT OFFSET  
The urgent offset field in the TCP header. If this value is not 0, the packet contains out-of-band (OOB) data.

TCP FLAG BYTE  
The flag byte in the TCP header. The numeric value is displayed along with the names of each bit that is set in the flag byte.

REASON CODE  
The reason code provided if an exception condition is associated with a packet.

IP HEADER  
The entire IP header of the packet.

TCP HEADER  
The entire TCP header of the packet.

DATA  
The user data in the packet that was traced.

- For IP packets using UDP, the following information is displayed:

RWI  
The read/write operation code, where:  
EVEN Even numbers represent read operations.  
ODD Odd numbers represent write operations.

IPCCW  
The IP channel command word (IPCCW) area index, where:  
01–C7 CDLC IP routers.  
D1–EE OSA-Express connections.  
FF Local sockets.

SOURCE IP  
The source IP address.

DEST IP

The destination IP address.

LEN

The length of the IP packet.

TOD

The time stamp.

PROTOCOL

The protocol of the IP packet, which for UDP is X'11' .

SOURCE PORT

The source port.

DEST PORT

The destination port.

IP HEADER

The entire IP header of the packet.

UDP HEADER

The entire UDP header of the packet.

DATA

The user data in the packet that was traced.

- For IP packets using a protocol other than TCP or UDP, the following information is displayed:

RWI

The read/write operation code, where:

EVEN Even numbers represent read operations.

ODD Odd numbers represent write operations.

IPCCW

The IP channel command word (IPCCW) area index, where:

01–C7 CDLC IP routers.

D1–EE

OSA-Express connections.

FF

Local sockets.

SOURCE IP

The source IP address.

DEST IP

The destination IP address.

LEN

The length of the IP packet.

TOD

The time stamp.

PROTOCOL

The protocol of the IP packet.

IP HEADER

The entire IP header of the packet.

DATA

The user data in the packet that was traced.

Figure 38 shows an example of a formatted IPTPRT report.

```
*****
          TRANSACTION PROCESSING FACILITY TCP/IP TRACE OUTPUT
*****
RECORDS MATCHING THE FOLLOWING SELECTION CRITERIA WILL BE PRINTED:
PROTOCOLS: . . . . . ALL
SOURCE PORTS: . . . . . ALL
DESTINATION PORTS: . . . . ALL
SOURCE IP ADDRESSES: . . . ALL
DESTINATION IP ADDRESSES: ALL
REASON CODES: . . . . . ALL
IP CCW: . . . . . ALL
DATE: . . . . . FROM JAN01 TO DEC31
TIME: . . . . . FROM 00:00:00 TO 23:59:59
TOD (FIRST WORD): . . . . FROM 00000000 TO FFFFFFFF
TCP FLAGS: . . . . . ALL
WIDE LAYOUT
IP FORMATTED TRACE
RWI-52 IPCCW-01 SOURCE IP-9.117.107.167 DEST IP-9.117.249.50 LEN-48
      TOD-B70C206B31D6FB20 PROTOCOL-06 (TCP) SOURCE PORT-1865 DEST PORT-21
      SEQ-102459496 WINDOW-16384 URGENT OFFSET-0
      TCP FLAG BYTE-02 (SYN)
      IP HEADER 45000030 7F254000 7D0606DF 09756BA7 0975F932
      TCP HEADER 07490015 061B6868 00000000 70024000 55DD0000 02040551 01010402
RWI-51 IPCCW-01 SOURCE IP-9.117.249.50 DEST IP-9.117.107.167 LEN-44
      TOD-B70C206B3254DA00 PROTOCOL-06 (TCP) SOURCE PORT-21 DEST PORT-1865
      SEQ-112401775 ACK-102459497 WINDOW-32767 URGENT OFFSET-0
      TCP FLAG BYTE-12 (ACK, SYN)
      IP HEADER 4500002C 9E070000 3C066901 0975F932 09756BA7
      TCP HEADER 00150749 06B31D6F 061B6869 60127FFF 06B20000 02040551
RWI-32 IPCCW-01 SOURCE IP-9.117.107.167 DEST IP-9.117.249.50 LEN-40
      TOD-B70C206B371FE785 PROTOCOL-06 (TCP) SOURCE PORT-1865 DEST PORT-21
      SEQ-102459497 ACK-112401776 WINDOW-17693 URGENT OFFSET-0
      TCP FLAG BYTE-10 (ACK)
      IP HEADER 45000028 7F264000 7D0606E6 09756BA7 0975F932
      TCP HEADER 07490015 061B6869 06B31D70 5010451D 58EE0000
```

Figure 38. Formatted IPTPRT Report

## IPTPRT Messages

When you submit the IPTPRT JCL to run the IPTPRT utility, the report will include a list of detected errors. For more information about the error message numbers shown in the report, see *Messages (System Error and Offline)* and *Messages (Online)*.

---

## Including the IP Trace Table in System Error Dumps

To include the IP trace table in system error dumps, use the ZIDOT command to specify the TCP/IP keyword (ITCP) in the dump override table. See *TPF Operations* for more information about the ZIDOT command.



---

## Appendix G. Management Information Base Variables

This appendix lists the variables defined by the Management Information Base (MIB) that are supported by the Simple Network Management Protocol (SNMP) agent on the TPF system. The MIB variable types are defined using the following fields:

### Variable Descriptor

A textual name for the variable.

### Object Identifier

The name for the variable in Abstract Syntax Notation (ASN.1) format.

---

## MIB Variable Types

The MIBs supported by the TPF system reside in core storage. The TPF system supports the MIB-II structure except for certain values (routing tables and physical address tables, for example) that do not apply to TPF. All MIBs supported by the TPF system are defined by the following Request for Comments (RFC) documents:

- RFC 1155 *Structure and Identification of Management Information for TCP/IP-based internets*
- RFC 1213 *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*
- RFC 2233 *The Interfaces Group MIB using SMIv2.*

Go to <http://www.ietf.org> for more information about these RFCs and any related extensions.

All MIB variables supported by TPF have read-only access. The following table shows the MIB-II variables defined by RFC 1213 that are supported by the TPF system. MIB variables that are not accessible are shown in ***bold italic*** font. The variables are listed in each SNMP group in the numeric order of their object identifiers.

Table 14. MIB Variables Supported by the TPF System

Variable Descriptor	Object Identifier
<b>System Group</b>	
sysDescr	1.3.6.1.2.1.1.1
sysObjectID	1.3.6.1.2.1.1.2
sysUpTime	1.3.6.1.2.1.1.3
sysContact	1.3.6.1.2.1.1.4
sysName	1.3.6.1.2.1.1.5
sysLocation	1.3.6.1.2.1.1.6
sysServices	1.3.6.1.2.1.1.7
<b>Interfaces Group</b>	
ifNumber	1.3.6.1.2.1.2.1
<b><i>ifTable</i></b>	1.3.6.1.2.1.2.2
ifEntry	1.3.6.1.2.1.2.2.1
ifIndex	1.3.6.1.2.1.2.2.1.1
ifDescr	1.3.6.1.2.1.2.2.1.2

Table 14. MIB Variables Supported by the TPF System (continued)

Variable Descriptor	Object Identifier
ifType	1.3.6.1.2.1.2.2.1.3
ifMtu	1.3.6.1.2.1.2.2.1.4
ifSpeed	1.3.6.1.2.1.2.2.1.5
ifPhysAddress	1.3.6.1.2.1.2.2.1.6
ifAdminStatus	1.3.6.1.2.1.2.2.1.7
ifOperStatus	1.3.6.1.2.1.2.2.1.8
ifLastChange	1.3.6.1.2.1.2.2.1.9
ifInOctets	1.3.6.1.2.1.2.2.1.10
ifInUcastPkts	1.3.6.1.2.1.2.2.1.11
ifInNUcastPkts	1.3.6.1.2.1.2.2.1.12
ifInDiscards	1.3.6.1.2.1.2.2.1.13
ifInErrors	1.3.6.1.2.1.2.2.1.14
ifInUnknownProtos	1.3.6.1.2.1.2.2.1.15
ifOutOctets	1.3.6.1.2.1.2.2.1.16
ifOutUcastPkts	1.3.6.1.2.1.2.2.1.17
ifOutNUcastPkts	1.3.6.1.2.1.2.2.1.18
ifOutDiscards	1.3.6.1.2.1.2.2.1.19
ifOutErrors	1.3.6.1.2.1.2.2.1.20
ifOutQLen	1.3.6.1.2.1.2.2.1.21
ifSpecific	1.3.6.1.2.1.2.2.1.22
<b>IP Group</b>	
ipForwarding	1.3.6.1.2.1.4.1
ipDefaultTTL	1.3.6.1.2.1.4.2
ipInReceives	1.3.6.1.2.1.4.3
ipInHdrErrors	1.3.6.1.2.1.4.4
ipInAddrErrors	1.3.6.1.2.1.4.5
ipForwDatagrams	1.3.6.1.2.1.4.6
ipInUnknownProtos	1.3.6.1.2.1.4.7
ipInDiscards	1.3.6.1.2.1.4.8
ipInDelivers	1.3.6.1.2.1.4.9
ipOutRequests	1.3.6.1.2.1.4.10
ipOutDiscards	1.3.6.1.2.1.4.11
ipOutNoRoutes	1.3.6.1.2.1.4.12
ipReasmTimeout	1.3.6.1.2.1.4.13
ipReasmReqds	1.3.6.1.2.1.4.14
ipReasmOKs	1.3.6.1.2.1.4.15
ipReasmFails	1.3.6.1.2.1.4.16
ipFragOKs	1.3.6.1.2.1.4.17
ipFragFails	1.3.6.1.2.1.4.18
ipFragCreates	1.3.6.1.2.1.4.19

Table 14. MIB Variables Supported by the TPF System (continued)

Variable Descriptor	Object Identifier
<b>ipAddrTable</b>	1.3.6.1.2.1.4.20
ipAddrEntry	1.3.6.1.2.1.4.20.1
ipAdEntAddr	1.3.6.1.2.1.4.20.1.1
ipAdEntIfIndex	1.3.6.1.2.1.4.20.1.2
ipAdEntNetMask	1.3.6.1.2.1.4.20.1.3
ipAdEntBcastAddr	1.3.6.1.2.1.4.20.1.4
ipAdEntReasmMaxSize	1.3.6.1.2.1.4.20.1.5
<b>ICMP Group</b>	
icmpInMsgs	1.3.6.1.2.1.5.1
icmpInErrors	1.3.6.1.2.1.5.2
icmpInDestUnreachs	1.3.6.1.2.1.5.3
icmpInTimeExcds	1.3.6.1.2.1.5.4
icmpInParmProbs	1.3.6.1.2.1.5.5
icmpInSrcQuenchs	1.3.6.1.2.1.5.6
icmpInRedirects	1.3.6.1.2.1.5.7
icmpInEchos	1.3.6.1.2.1.5.8
icmpInEchoReps	1.3.6.1.2.1.5.9
icmpInTimestamps	1.3.6.1.2.1.5.10
icmpInTimestampReps	1.3.6.1.2.1.5.11
icmpInAddrMasks	1.3.6.1.2.1.5.12
icmpInAddrMaskReps	1.3.6.1.2.1.5.13
icmpOutMsgs	1.3.6.1.2.1.5.14
icmpOutErrors	1.3.6.1.2.1.5.15
icmpOutDestUnreachs	1.3.6.1.2.1.5.16
icmpOutTimeExcds	1.3.6.1.2.1.5.17
icmpOutParmProbs	1.3.6.1.2.1.5.18
icmpOutSrcQuenchs	1.3.6.1.2.1.5.19
icmpOutRedirects	1.3.6.1.2.1.5.20
icmpOutEchos	1.3.6.1.2.1.5.21
icmpOutEchoReps	1.3.6.1.2.1.5.22
icmpOutTimestamps	1.3.6.1.2.1.5.23
icmpOutTimestampReps	1.3.6.1.2.1.5.24
icmpOutAddrMasks	1.3.6.1.2.1.5.25
icmpOutAddrMaskReps	1.3.6.1.2.1.5.26
<b>TCP Group</b>	
tcpRtoAlgorithm	1.3.6.1.2.1.6.1
tcpRtoMin	1.3.6.1.2.1.6.2
tcpRtoMax	1.3.6.1.2.1.6.3
tcpMaxConn	1.3.6.1.2.1.6.4
tcpActiveOpens	1.3.6.1.2.1.6.5

Table 14. MIB Variables Supported by the TPF System (continued)

Variable Descriptor	Object Identifier
tcpPassiveOpens	1.3.6.1.2.1.6.6
tcpAttemptFails	1.3.6.1.2.1.6.7
tcpEstabResets	1.3.6.1.2.1.6.8
tcpCurrEstab	1.3.6.1.2.1.6.9
tcpInSegs	1.3.6.1.2.1.6.10
tcpOutSegs	1.3.6.1.2.1.6.11
tcpRetransSegs	1.3.6.1.2.1.6.12
<b>tcpConnTable</b>	1.3.6.1.2.1.6.13
tcpConnEntry	1.3.6.1.2.1.6.13.1
tcpConnState	1.3.6.1.2.1.6.13.1.1
tcpConnLocalAddress	1.3.6.1.2.1.6.13.1.2
tcpConnLocalPort	1.3.6.1.2.1.6.13.1.3
tcpConnRemAddress	1.3.6.1.2.1.6.13.1.4
tcpConnRemPort	1.3.6.1.2.1.6.13.1.5
tcpInErrs	1.3.6.1.2.1.6.14
tcpOutRsts	1.3.6.1.2.1.6.15
<b>UDP Group</b>	
udpInDatagrams	1.3.6.1.2.1.7.1
udpNoPorts	1.3.6.1.2.1.7.2
udpInErrors	1.3.6.1.2.1.7.3
udpOutDatagrams	1.3.6.1.2.1.7.4
<b>udpTable</b>	1.3.6.1.2.1.7.5
udpEntry	1.3.6.1.2.1.7.5.1
udpLocalAddress	1.3.6.1.2.1.7.5.1.1
udpLocalPort	1.3.6.1.2.1.7.5.1.2
<b>SNMP Group</b>	
snmpInPkts	1.3.6.1.2.1.11.1
snmpOutPkts	1.3.6.1.2.1.11.2
snmpInBadVersions	1.3.6.1.2.1.11.3
snmpInBadCommunityNames	1.3.6.1.2.1.11.4
snmpInBadCommunityUses	1.3.6.1.2.1.11.5
snmpInASNParseErrs	1.3.6.1.2.1.11.6
NOT USED	1.3.6.1.2.1.11.7
snmpInTooBigs	1.3.6.1.2.1.11.8
snmpInNoSuchNames	1.3.6.1.2.1.11.9
snmpInBadValues	1.3.6.1.2.1.11.10
snmpInReadOnlys	1.3.6.1.2.1.11.11
snmpInGenErrs	1.3.6.1.2.1.11.12
snmpInTotalReqVars	1.3.6.1.2.1.11.13
snmpInTotalSetVars	1.3.6.1.2.1.11.14

Table 14. MIB Variables Supported by the TPF System (continued)

<b>Variable Descriptor</b>	<b>Object Identifier</b>
snmpInGetRequests	1.3.6.1.2.1.11.15
snmpInGetNexts	1.3.6.1.2.1.11.16
snmpInSetRequests	1.3.6.1.2.1.11.17
snmpInGetResponses	1.3.6.1.2.1.11.18
snmpInTraps	1.3.6.1.2.1.11.19
snmpOutTooBig	1.3.6.1.2.1.11.20
snmpOutNoSuchNames	1.3.6.1.2.1.11.21
snmpOutBadValues	1.3.6.1.2.1.11.22
NOT USED	1.3.6.1.2.1.11.23
snmpOutGenErrs	1.3.6.1.2.1.11.24
snmpOutGetRequests	1.3.6.1.2.1.11.25
snmpOutGetNexts	1.3.6.1.2.1.11.26
snmpOutSetRequests	1.3.6.1.2.1.11.27
snmpOutGetResponses	1.3.6.1.2.1.11.28
snmpOutTraps	1.3.6.1.2.1.11.29
snmpEnableAuthenTraps	1.3.6.1.2.1.11.30



---

# Index

## Special Characters

- /etc/imapd.conf 259
- /etc/postfix/access 259
- /etc/postfix/main.cf 259
- /etc/syslog.conf 246
- /etc/syslog.pid 245
- /etc/tftp.conf 239, 241
- /etc/tpf\_mail.conf 259
- # directive 240
- #MAILxx record 255
- accept socket function
  - accept a connection request 138
- activate\_on\_accept socket function 82
  - activate a program after data received 141
- activate\_on\_receipt function call, using 129
- activate\_on\_receipt function, sample child server code 369
- activate\_on\_receipt function, sample server code 367
- activate\_on\_receipt socket function
  - activate a program after data received 144
  - activate a program after data received with length 148
- bind socket function
  - bind a local name to the socket 152
- claw\_accept function 386
- claw\_closeadapter function 388
- claw\_connect function 390
- claw\_disconnect function 393
- claw\_end function 395
- claw\_initialization function 396
- claw\_openadapter function 397
- claw\_query function 400
- claw\_send function 403
- close socket function
  - shut down a socket 155
- connect socket function
  - request a connection to a remote host 157
- gethostbyaddr socket function
  - get host information by address 160
- gethostbyname socket function
  - get host information by name 162
- gethostid socket function
  - return host identifier 164
- gethostname socket function
  - return the host name 166
- getpeername socket function
  - return the name of the peer 168
- getservbyname socket function
  - get server port by name 170
- getservbyport socket function
  - get server name by port 172
- getsockname socket function
  - return the name of the local socket 174
- getsockopt socket function
  - return socket options 176
- htonl socket function
  - translate a long integer 180
- htons socket functions
  - translate a short integer 181
- inet\_addr socket function
  - construct internet address from character string 182
- inet\_ntoa socket function
  - return pointer to a string in dotted decimal notation 184
- ioctl socket function
  - perform special operations on socket 185
- listen socket function
  - complete binding 189
- ntohl socket function
  - translate a long integer 191
- ntohs socket function
  - translate a short integer 192
- read socket function
  - read data on a socket 193
- recv socket function
  - receive data on a connected socket 196
- recvfrom socket function
  - receive data on connected/unconnected socket 199
- recvmsg socket function
  - receive messages on a socket 202
- select socket function
  - monitor read, write, and exception status 205
- send socket function
  - send data on a connected socket 206
- sendmsg socket function
  - send message on a socket 210
- sendto socket function
  - send data on an unconnected socket 212
- setsockopt socket function
  - set options associated with a socket 216
- shutdown socket function
  - shut down all or part of a duplex connection 220
- sock\_errno socket function
  - return the error code set by a socket call 222
- socket.h header file 363
- socket socket function
  - create an endpoint for communication 223
- SSL\_accept SSL function
  - accept an SSL session connection request 274
- SSL\_aor SSL function
  - allow the issuing ECB to exit and a specific program to be activated in a new ECB 275
- SSL\_check\_private\_key SSL function
  - verify private key-public key agreement 277
- SSL\_connect SSL function
  - start an SSL session 279
- SSL\_CTX\_check\_private\_key SSL function
  - verify private key- public key agreement in the certificate 280
- SSL\_CTX\_free SSL function
  - return a context (CTX) structure to the system 281

**SSL\_CTX\_load\_and\_set\_client\_CA\_list** SSL function  
 load certificates from a file and places the issuer name of each certificate in a CTX structure 282

**SSL\_CTX\_load\_verify\_locations** SSL function  
 load the certificate authorities (CAs) 283

**SSL\_CTX\_new\_shared** SSL function  
 create a new CTX structure for use by shared SSL sessions 287

**SSL\_CTX\_new** SSL function  
 create a new context (CTX) structure 285

**SSL\_CTX\_set\_cipher\_list** SSL function  
 set the cipher list for use by SSL sessions 289

**SSL\_CTX\_set\_client\_CA\_list** SSL function  
 identify set of CAs sent to remote client 292

**SSL\_CTX\_set\_default\_passwd\_cb\_userdata** SSL function  
 identify the password to access data in a private key in PEM format 293

**SSL\_CTX\_set\_verify** SSL function  
 indicate whether to verify remote peers starting SSL sessions 294

**SSL\_CTX\_use\_certificate\_chain\_file** SSL function  
 load the chain of certificates for an SSL session to use in a specific context 296

**SSL\_CTX\_use\_certificate\_file** SSL function  
 load the certificate for an SSL session to use in a specific context 298

**SSL\_CTX\_use\_PrivateKey\_file** SSL function  
 load the private key for an SSL session to use in a specific context 300

**SSL\_CTX\_use\_RSAPrivateKey\_file** SSL function  
 load the RSA private key for an SSL session to use in a specific context 302

**SSL\_free** SSL function  
 returns to the system the SSL structure associated with an SSL session 304

**SSL\_get\_cipher** SSL function  
 returns the cipher name associated with a specific SSL session 305

**SSL\_get\_error** SSL function  
 return error information about an SSL API 307

**SSL\_get\_peer\_certificate** SSL function  
 return the peer certificate received from an SSL session 309

**SSL\_get\_session** SSL function  
 return a copy of the SSL session information for a specific SSL structure 310

**SSL\_get\_verify\_result** SSL function  
 return the result of the remote peer certificate validation 311

**SSL\_get\_version** SSL function  
 returns the protocol version of the current SSL connection 313

**SSL\_library\_init** SSL function  
 registers the available ciphers and message digest 314

**SSL\_load\_and\_set\_client\_CA\_list** SSL function  
 load certificates from a file and puts the name of each certificate in the SSL structure 315

**SSL\_load\_client\_CA\_file** SSL function  
 load certificates from a file 316

**SSL\_new** SSL function  
 create a new SSL structure for use by an SSL session 317

**SSL\_pending** SSL function  
 return data in the current SSL data record that is available for reading on an SSL session 318

**SSL\_read** SSL function  
 read application data from an SSL session 319

**SSL\_renegotiate** SSL function  
 create a new set of cipher keys for an existing SSL session 320

**SSL\_set\_cipher\_list** SSL function  
 set the ciphers for use by an SSL session 321

**SSL\_set\_client\_CA\_list** SSL function  
 identify a CA list for use with client certificate requests 324

**SSL\_set\_fd** SSL function  
 assign a socket to an SSL structure 325

**SSL\_set\_session** SSL function  
 set up SSL session information when reusing an SSL session 326

**SSL\_set\_verify** SSL function  
 indicate whether to verify the remote client identity when an SSL session starts 327

**SSL\_shutdown** SSL function  
 shut down data flow for an SSL session 329

**SSL\_use\_certificate\_file** SSL function  
 load the certificate for use with an SSL session 330

**SSL\_use\_PrivateKey\_file** SSL function  
 load the private key for use with an SSL session 332

**SSL\_use\_RSAPrivateKey\_file** SSL function  
 load the RSA private key for use with an SSL session 333

**SSL\_write** SSL function  
 write application data across an SSL session 334

**SSLv2\_client\_method** SSL function  
 indicate that an application is a client and supports SSL version 2 335

**SSLv2\_server\_method** SSL function  
 indicate that an application is a server and supports SSL version 2 336

**SSLv23\_client\_method** SSL function  
 indicate that an application is a client and supports SSL versions 2 and 3 and TLS version 1 337

**SSLv23\_server\_method** SSL function  
 indicate that an application is a server and supports SSL versions 2 and 3 and TLS version 1 338

**SSLv3\_client\_method** SSL function  
 indicate that an application is a client and supports SSL version 3 339

**SSLv3\_server\_method** SSL function  
 indicate that an application is a server and supports SSL version 3 340

**TLSv1\_client\_method** SSL function  
 indicate that an application is a client and supports TLS version 1 341

**TLSv1\_server\_method**  
 indicate that an application is a server and supports TLS version 1 342

**tpf\_vipac** C function use for moving a VIPA 69

- write socket function
  - write data on a connected socket 226
- writew socket function
  - write data on a connected socket 229

## Numerics

3172 Model 3 Interconnect Controller 30

## A

- about the IP trace table 407
- accept a connect request from CLAW workstation 386
- accept a connection request
  - socket API function, accept 138
- accept an SSL session connection request
  - SSL API function, SSL\_accept 274
- access control list (ACL) 256
- access list 259, 264
  - /etc/postfix/access 259
  - creating 259
  - parameters 264
- access the Internet, application programs
  - socket API functions 137
- activate a CLAW logical link request 390
- activate a program after data received
  - socket API function, activate\_on\_accept 141
  - socket API function,
    - activate\_on\_receipt\_with\_length 148
    - socket API function, activate\_on\_receipt 144
- activating and deactivating CDLC IP routers 65
- activating and deactivating OSA-Express connections 67
- ACTIVATION parameter 237
- active logical link, send message on 403
- active queue 257
- active queue control record 257
- active queue record 257
- adapter, query status 400
- adapter, remove logical link 393
- adding entry for an Internet server application to the IDCF 235
- adding FTP server entry to the IDCF 243
- adding IP routing table entries 74
- adding syslog daemon server entry to the IDCF 249
- adding TFTP server entry to the IDCF 239
- address parts
  - mapping 120
- allow directive 240
- allows the issuing ECB to exit and a specific program to be activated in a new ECB
  - SSL API function, SSL\_aor 275
- API (application programming interface) 137
- API support, socket 122
- application program use for moving a VIPA 69
- application programming interface (API) 137
- application programming interface functions
  - socket application 137
- assign a socket to an SSL structure
  - SSL API function, SSL\_set\_fd 325
- AUTH directive 240

## B

- balancing workload with movable VIPAs 69
- basic encoding rules (BER) and SNMP 91
- begin activity on a CLAW logical link request 390
- bind a local name to the socket
  - socket API function, bind 152
- blocking mode, socket
  - wait for data 121
- buffer sizes, send and receive 80
- byte order conversion
  - integer 120

## C

- CDLC IP CCW area table 52
- CDLC IP CCW area table resources, defining 52
- CDLC IP configuration record 51
- CDT 23
- channel data link protocol (CDLC) 40
- child server code, sample
  - using activate\_on\_receipt function 369
- CLAW activity on subchannel pair, terminate 388
- CLAW device table (CDT) 23
  - CLAWADP value 23
  - updated 23
- CLAWADP value, choosing 23
- CLAWFD value, choosing 25
- CLAWIP value, choosing 26
- client code, sample
  - TCP client 373
  - UDP client 378
- client support 122
- client/server environment 7
- close inactive sockets
  - socket sweeper 125
- coding the SNMP user exits 98
- commands
  - ZCLAW ACTIVATE 30
  - ZCLAW ADD 29
  - ZCLAW DELETE 31
  - ZCLAW DISPLAY 31
  - ZCLAW INACTIVATE 30
  - ZCLAW RESET 33
  - ZCLAW TRACE 32, 33
  - ZINET ADD 99, 237
  - ZINET ALTER 235, 237
  - ZINET DELETE 235
  - ZINET START 99, 237, 238
  - ZINET STOP 237, 238
  - ZOSAE 63, 64, 65, 67, 68
  - ZSNMP 98
  - ZTRTE 99
  - ZTTCP DISPLAY 67
  - ZVIPA 65, 68, 69, 70, 71
- communication endpoint
  - socket 119
- complete binding
  - socket API function, listen 189
- components
  - socket/CLAW interfaces 16

- components (*continued*)
  - TCP/IP native stack support 44
- concepts used in data transmission
  - socket 119, 120
- configuration file
  - access list
    - general discussion 259
    - parameters 264
  - IMAP/POP
    - general discussion 259
    - parameters 262
  - SMTP
    - general discussion 259
    - parameters 259
  - syslog daemon
    - updating 249
  - TFTP
    - creating 241
    - directives 240
    - general discussion 239
    - transferring to TPF system 241
    - updating 241
  - TPF
    - general discussion 259
    - parameters 263
- configuration file with SNMP
  - creating 96
  - example 96
  - keywords 97
  - refresh with ZSNMP command 96
- configuration options
  - access list 264
  - IMAP/POP configuration file 262
  - SMTP configuration file 259
  - TPF configuration file 263
- configuring a TPF system 59
- connect request from CLAW workstation, accept 386
- connect request from the 3172 Model 3, accept 386
- construct internet address from character string
  - socket API function, `inet_addr` 182
- control blocks
  - CLAW device table (CDT) 23
  - file descriptor table (FDT) 24
  - internet protocol address table (IPT) 25
- conversion
  - integer byte order 120
- create a new context (CTX) structure
  - SSL API function, `SSL_CTX_new` 285
- create a new SSL structure for use by an SSL session
  - SSL API function, `SSL_new` 317
- create an endpoint for communication
  - socket API function, `socket` 223
- creates a new CTX structure for use by shared SSL sessions.
  - SSL API function, `SSL_CTX_new_shared` 287
- creates a new set of cipher keys for an existing SSL session
  - SSL API function, `SSL_renegotiate` 320
- creating a compacted display of the IP trace table 410
- creating a formatted display of the IP trace table 410
- creating the SNMP configuration file 96

- creating the TFTP configuration file 241
- creating the TPF Internet mail server configuration files 259

## D

- data
  - out-of-band 121
- data collection and reduction 112
- data flow
  - between the IP router and the TPF System 41
  - between the OSA-Express card and the TPF System 44
  - TPF system and IBM 3172 Model 3 Interconnect Controller 12
- data trace postprocessor, CLAW output 344
- data trace postprocessor, sample JCL 343
- data transmission
  - socket concepts used 120
- datagram sockets
  - User Datagram Protocol (UDP) 119
- deactivating sockets 77
- deactivation processing within movable VIPA support
  - and user exit 69
- default local IP address 61
- deferred queue 257
- deferred queue control record 257
- deferred queue record 257
- defining
  - CDLC IP CCW area table resources 52
  - CDLC IP network configuration 52
  - CDLC IP routers 65
  - CDLC local IP addresses 62
  - gateways 58
  - IP message table 54
  - IP routing table 54
  - local IP addresses 60
  - OSA read buffers 53
  - OSA-Express cards 66
  - OSA-Express connections 67
  - real OSA IP addresses 63
  - routing table entries for SNMP 99
  - SNMP agent server 98
  - socket block table 51
  - TPF system to SNMP 99
  - VIPAs 64
- deleting
  - CDLC IP routers 66
  - CDLC local IP addresses 62
  - IP routing table entries 74
  - OSA-Express connections 67
  - real OSA IP addresses 63
  - VIPAs 65
- denial-of-service attacks 105
- deny directive 240
- describing SNMP agent support 89
- description of a sample TCP/IP network 13
- differentiated services 47, 111
- directives
  - # 240, 248
  - allow 240

- directives (*continued*)
  - AUTH 240
  - deny 240
  - LOG 240
  - syslog daemon configuration file 246
  - TFTP configuration file 240
- disconnect a logical link from an adapter 393
- dispatch a message on an active logical link 403
- displaying
  - individual IP trace tables 77
  - IP routing table entries 74
  - IP trace information 75
  - OSA-Express connections 67
  - socket control block information 78
  - TCP/IP native stack support 75
  - VIPA statistics 69
  - VIPAs 65
- displaying information about the IP trace facility 409
- displaying the IP trace table online 409
- DNS client 103
- DNS server 101
- Domain Name System (DNS) Support
  - DNS client 103
  - DNS server 101
  - IP address selection 102
  - TPF host name table 101
- dotted decimal formats 120
  - standard 120

## E

- ECBs
  - sockets 119
- enabling TCP/IP native stack support 60
- end all CLAW activity 395
- end CLAW activity on subchannel pair 388
- enterprise-specific SNMP traps 93
- enterprise-specific user MIB variables and SNMP 93

## F

- FDT 24
- file descriptor table (FDT) 24
  - CLAWFD value 25
  - updated 24
- File Transfer Protocol (FTP) server
  - adding entry to IDCf 243
  - general discussion 243
- FTP LOG file 243
- full-duplex socket support
  - chained 124
  - socket thread control blocks 124

## G

- gateways 68
- gateways, defining 58
- get host information by address
  - socket API function, `gethostbyaddr` 160
- get host information by name
  - socket API function, `gethostbyname` 162

- get server port by name
  - socket API function, `getservbyname` 170
  - socket API function, `getservbyport` 172
- get the status of CLAW adapter 400
- get the status of CLAW logical links 400

## I

- IBM 3172 Model 3 Interconnect Controller
  - installation 29
- identify a CA list for use with client certificate requests
  - SSL API function, `SSL_set_client_CA_list` 324
- identify set of CAs sent to remote client
  - SSL API function, `SSL_CTX_set_client_CA_list` 292
- identify the password to access data in a private key in PEM format
  - SSL API function,
    - `SSL_CTX_set_default_passwd_cb_userdata` 293
- IMAP server
  - general discussion 253
- IMAP/POP configuration file
  - `/etc/imapd.conf` 259
  - creating 259
  - general discussion 259
  - parameters 262
- inbound message flow, TCP/IP native stack support 48
- indicate that an application is a client and supports SSL version 2
  - SSL API function, `SSLv2_client_method` 335
- indicate that an application is a client and supports SSL version 3
  - SSL API function, `SSLv3_client_method` 339
- indicate that an application is a client and supports SSL versions 2 and 3 and TLS version 1
  - SSL API function, `SSLv23_client_method` 337
- indicate that an application is a client and supports TLS version 1
  - SSL API function, `TLSv1_client_method` 341
- indicate that an application is a server and supports SSL version 2
  - SSL API function, `SSLv2_server_method` 336
- indicate that an application is a server and supports SSL version 3
  - SSL API function, `SSLv3_server_method` 340
- indicate that an application is a server and supports SSL versions 2 and 3 and TLS version 1
  - SSL API function, `SSLv23_server_method` 338
- indicate that an application is a server and supports TLS version 1
  - SSL API function, `TLSv1_server_method` 342
- indicate whether to verify remote peers starting SSL sessions
  - SSL API function, `SSL_CTX_set_verify` 294
- indicate whether to verify the remote client identity when an SSL session starts
  - SSL API function, `SSL_set_verify` 327
- individual IP trace support
  - configuring 59
  - displaying individual tables 77
  - using 76
- industry standards 7

- initialize a CLAW adapter 397
- initialize CLAW activity 396
- initiate a request to open a CLAW logical link 390
- integer byte order conversion 120
- interconnection of networks 3, 13
- interface
  - socket APIs 137
- interfaces, socket/CLAW
  - inbound message flow 18
  - inbound message flow through 18
  - outbound message flow 17
- internals, TCP/IP native stack support 39
- Internet daemon
  - listener 236
  - monitor 236
  - operational considerations 235, 236
  - operator procedures 235
  - starting 237
  - stopping 237
- Internet daemon configuration file (IDCF)
  - adding an entry for the TPF Internet mail servers 267
  - adding entry for an Internet server application 235
  - adding entry for the File Transfer Protocol (FTP) server 243
  - adding entry for the syslog daemon server 249
  - adding entry for the Trivial File Transfer Protocol (TFTP) server 239
  - general discussion 235
- Internet mail
  - receiving 269
  - sending 269
- Internet Protocol (IP) trace facility, using 407
- internet protocol address table (IPT) 25
  - CLAWIP value 26
  - updated 25
- Internet security, overview 105
- Internet server application
  - ACTIVATION parameter 237
  - IP parameter 235
  - operator control 238
  - parameters in ZINET ADD command 237
  - parameters in ZINET ALTER command 237
  - STATE parameter 238
  - USER parameter 237
- IP address selection 102
- IP message table 53
- IP message table, defining 54
- IP parameter 235, 236
- IP protocol
  - raw sockets 119
- IP routers, configuration characteristics 41
- IP routing table 54
- IP trace facility
  - defining the IPTPRT report 411
  - displaying information about 409
  - individual IP trace support 76
  - IP trace table
    - creating a compacted display 410
    - creating a formatted display 410
    - displaying online 409, 421

- IP trace facility (*continued*)
  - IP trace table (*continued*)
    - information about 407
    - storing data 408
    - writing to a real-time tape 409
  - IPTPRT utility 410
  - IPTPRT utility messages 421
  - sample JCI for the IPTPRT utility 411
  - starting 407
  - stopping 408
  - using 407
- IP trace facility, using 407
- IPT 25
- IPTPRT utility 410, 421
- ISO-C structures
  - socket API functions, use 363

## J

- JCL, data trace postprocessor 343
- JCL, process data trace postprocessor 350

## L

- LAN 3
- load certificates from a file
  - SSL API function, `SSL_load_client_CA_file` 316
- load the certificate authorities (CAs)
  - SSL API function,
    - `SSL_CTX_load_verify_locations` 283
- load the certificate for an SSL session to use in a specific context
  - SSL API function,
    - `SSL_CTX_use_certificate_file` 298
- load the certificate for use with an SSL session
  - SSL API function, `SSL_use_certificate_file` 330
- load the chain of certificates for an SSL session to use in a specific context
  - SSL API function,
    - `SSL_CTX_use_certificate_chain_file` 296
- load the private key for an SSL session to use in a specific context
  - SSL API function,
    - `SSL_CTX_use_PrivateKey_file` 300
- load the private key for use with an SSL session
  - SSL API function, `SSL_use_PrivateKey_file` 332
- load the RSA private key for an SSL session to use in a specific context
  - SSL API function,
    - `SSL_CTX_use_RSAPrivateKey_file` 302
- load the RSA private key for use with an SSL session
  - SSL API function, `SSL_use_RSAPrivateKey_file` 333
- loads certificates from a file and places the issuer name of each certificate in a CTX structure
  - SSL API function,
    - `SSL_CTX_load_and_set_client_CA_list` 282
- loads certificates from a file and puts the name of each certificate in the SSL structure
  - SSL API function,
    - `SSL_load_and_set_client_CA_list` 315
- local sockets 82

- LOG directive 240
- log files 245
  - offloading 251
- logging rules 246
- logical link on CLAW device, open request 390
- logical link, remove from adapter 393
- logical link, send message on 403
- logical links, query status 400
- low-water marks 81

## M

- mail database
  - #MAILxx record 255, 258
  - access control list (ACL) 256
  - active queue 257
  - deferred queue 257
  - recoup considerations 258
  - user profile record (UPR) 255
- mail queue
  - active 257
  - deferred 257
- mailbox, managing 269
- Management Information Base (MIB)
  - description with SNMP 85, 90
  - enterprise-specific, SNMP 93
  - protocol groups with SNMP 90
  - variables table 423
- mapping address parts 120
- maximum packet size 61
- message counters 112
- message flow through the socket/CLAW interfaces
  - inbound 18
  - outbound 17
- message on an active logical link, send 403
- messages, IPTPRt utility 421
- MIB variable table 423
- MIB variables and SNMP 85, 92, 93
- modifying IP routing table entries 74
- modifying the syslog daemon configuration file 249
- monitor read, write, and exception status
  - socket API function, `select` 205
- movable VIPAs
  - and deactivation processing 69
  - and network traffic balancing 69
  - and workload balancing 69
  - defined 64
  - moving by application program 69
  - moving to another processor 68
- moving a VIPA by application program 69
- moving VIPAs to another processor 68
- MSG\_OOB flag
  - out-of-band data, set for 121

## N

- network priority 111
- network protocols
  - introduction of 3
- network requirements
  - client/server environment 7

- network requirements (*continued*)
  - industry standard 7
  - open network connectivity 7
  - porting socket applications 7
  - role in the Internet 7
- network traffic and movable VIPAs 69
- network, computer 3
- network, interconnection 3, 13
- network, tuning 55
- nonblocking mode, socket
  - do not wait for data 121

## O

- obtain the status of CLAW adapter 400
- obtain the status of CLAW logical links 400
- offload device
  - IBM 3172 Model 3 Interconnect Controller 29
- open a CLAW adapter 397
- open a CLAW logical link request 390
- open network connectivity 7
- operational considerations
  - File Transfer Protocol (FTP) server 243
  - Internet daemon 235, 236
  - Internet server application 237
  - syslog daemon 245
  - Trivial File Transfer Protocol (TFTP) server 239
- operator control
  - Internet daemon 236
  - Internet server application 238
- order conversion
  - integer byte 120
- OSA configuration record 52
- OSA control block table 52
- OSA read buffers 53
- OSA read buffers, defining 53
- OSA shared IP address table (OSIT) 52
- OSA-Express card 39
- OSA-Express card, configuration characteristics 43
- OSA-Express connections 66
- OSA-Express support 5, 43
- out-of-band data
  - MSG\_OOB flag 121
- outbound message flow through the socket/CLAW interfaces 17
- outbound message flow, OSA-Express support 48
- outbound message flow, TCP/IP native stack support 46

## P

- packet filtering 105
  - `/etc/iprules.txt` 105, 106
  - defining rules for 106
- perform special operations on socket
  - socket API function, `ioctl` 185
- performance, TCP/IP native stack support 56
- policy agent, TCP/IP native stack support 46
- POP server
  - general discussion 253
- port 120

- porting socket applications 7
- prepare a CLAW adapter 397
- prepare for CLAW activity 396
- process model
  - DAEMON 236
  - NOLISTEN 236
  - RPC 236
- process trace postprocessor 350
- process trace postprocessor, sample JCL 350
- processing SNMP Requests 91
- protocol data units (PDU)
  - format table 88, 91
  - structure and fields, SNMP 87
  - trap format table 88
  - variable binding table 89

## Q

- quality of service (QoS) 47, 111
- query the status of CLAW adapter 400
- query the status of CLAW logical links 400
- queued direct I/O (QDIO) 40, 43

## R

- raw sockets
  - IP protocol 119
- read application data from an SSL session
  - SSL API function, SSL\_read 319
- read data on a socket
  - socket API function, read 193
- real OSA IP addresses 63
- receive a connect request from CLAW workstation 386
- receive data on a connected socket
  - socket API function, recv 196
- receive data on a connected/unconnected socket
  - socket API function, recvfrom 199
- receive messages on a socket
  - socket API function, recvmsg 202
- recoup
  - BKD1 258
  - updating descriptors for a mail database 258
- refresh SNMP configuration file with ZSNMP
  - command 98
- register the available ciphers and digest
  - SSL API function, SSL\_library\_init 314
- remove a logical link from an adapter 393
- removing VIPAs 65
- request a connection to a remote host
  - socket API function, connect 157
- request the status of CLAW adapter 400
- request the status of CLAW logical links 400
- restricted C functions
  - TCP/IP 385
- restricted CDLC IP address 63
- restricted functions
  - claw\_accept 386
  - claw\_closeadapter 388
  - claw\_connect 390
  - claw\_disconnect 393
  - claw\_end 395

- restricted functions (*continued*)
  - claw\_initialization 396
  - claw\_openadapter 397
  - claw\_query 400
  - claw\_send 403
- return a context (CTX) structure to the system
  - SSL API function, SSL\_CTX\_free 281
- return data in the current SSL data record that is available for reading on an SSL session
  - SSL API function, SSL\_pending 318
- return error information about an SSL API
  - SSL API function, SSL\_get\_error 307
- return pointer to a string in dotted decimal notation
  - socket API function, inet\_ntoa 184
- return socket options
  - socket API function, getsockopt 176
- return the cipher name associated with a specific SSL session
  - SSL API function, SSL\_get\_cipher 305
- return the error code set by a socket call
  - socket API function, sock\_errno 222
- return the name of the local socket
  - socket API function, getsockname 174
- return the name of the peer
  - socket API function, getpeername 168
- return the offload device name
  - socket API function, gethostname 166
- return the peer certificate received from an SSL session
  - SSL API function, SSL\_get\_peer\_certificate 309
- return the protocol version of the current SSL connection
  - SSL API function, SSL\_get\_version 313
- return the result of the remote peer certificate validation
  - SSL API function, SSL\_get\_verify\_result 311
- return to the system the SSL structure associated with an SSL session
  - SSL API function, SSL\_free 304
- returns a copy of the SSL session information for a specific SSL structure
  - SSL API function, SSL\_get\_session 310
- returns host identifier
  - socket API function, gethostid 164
- RIP 68
- role in the Internet 7
- routing information protocol (RIP) 68
- routing table entries and SNMP 99

## S

- sample TCP/IP network
  - description 13
  - TPF system connected to one IP network 41
- Secure Sockets Layer (SSL) support 273
- send a message on an active logical link 403
- send data on a connected socket
  - socket API function, send 206
- send data on an unconnected socket
  - socket API function, sendto 212
- send messages on a socket
  - socket API function, sendmsg 210

- server code, sample
  - TCP server 371
  - UDP server 376
  - using `activate_on_receipt` function, 367
- server, TFTP 239
- set options associated with a socket
  - socket API function, `setsockopt` 216
- set the cipher list for use by SSL sessions
  - SSL API function, `SSL_CTX_set_cipher_list` 289
- set the ciphers for use by an SSL session
  - SSL API function, `SSL_set_cipher_list` 321
- sets up SSL session information when reusing an SSL session
  - SSL API function, `SSL_set_session` 326
- sharing sockets 79
- shut down a socket
  - socket API functions, `close` 155
- shut down all or part of a duplex connection
  - socket API function, `shutdown` 220
- shut down data flow for an SSL session
  - SSL API function, `SSL_shutdown` 329
- Simple Network Management Protocol (SNMP) agent support, overview 85
- SMTP configuration file
  - `/etc/postfix/main.cf` 259
  - creating 259
  - general discussion 259
  - parameters 259
- SMTP server
  - general discussion 253
- SNAKEY parameters
  - CLAWADP 23
  - CLAWFD 25
  - CLAWIP 26
  - SOCKSWP 125
- SNMP agent support
  - agent description 85
  - component diagrams 86
  - configuration file 96
  - configuration file keywords 97
  - description 89
  - installation 96
  - installation of TCP/IP native stack support 96
  - manager description 85
  - message processing 91
  - MIB description 85
  - MIB variables table 423
  - MIB-II description 90
  - overview 85
  - processing requests 91
  - protocol data units (PDUs) 87
  - routing table entries 99
  - server definition 98
  - TPF system definition 99
  - traps 93
  - user exits 98
  - user MIB variables 93
- SNMP traps
  - example 93
  - types 93
- socket
  - communication endpoint 119
  - socket address, Internet 119
  - socket API functions
    - `accept` 138
    - `activate_on_accept` 141
    - `activate_on_receipt_with_length` 148
    - `activate_on_receipt` 144
    - `bind` 152
    - `close` 155
    - `connect` 157
    - `gethostbyaddr` 160
    - `gethostbyname` 162
    - `gethostid` 164
    - `gethostname` 166
    - `getpeername` 168
    - `getservbyname` 170
    - `getservbyport` 172
    - `getsockname` 174
    - `getsockopt` 176
    - `htonl` 180
    - `htons` 181
    - `inet_addr` 182
    - `inet_ntoa` 184
    - `ioctl` 185
    - `listen` 189
    - `ntohl` 191
    - `ntohs` 192
    - `read` 193
    - `recv` 196
    - `recvfrom` 199
    - `recvmsg` 202
    - `select` 205
    - `send` 206
    - `sendmsg` 210
    - `sendto` 212
    - `setsockopt` 216
    - `shutdown` 220
    - `sock_errno` 222
    - `socket` 223
    - `write` 226
    - `writew` 229
  - socket API functions, using
    - access the Internet 137
  - socket API support 122
  - socket block table structure 51
  - socket calls
    - using 132
  - socket concepts used in data transmission 120
  - socket nonblocking mode
    - do not wait for data 121
  - socket options supported, TCP/IP native stack support 80
  - socket sweeper
    - close inactive sockets 125
  - socket thread control blocks
    - chained 124
    - full-duplex socket support 124
  - socket user exits 122
  - socket/CLAW interfaces
    - inbound message 18

- socket/CLAW interfaces (*continued*)
  - inbound message flow through 18
  - outbound message 17
  - outbound message flow through 17
- socket/CLAW interfaces, TCP/IP support components 16
- sockets
  - closing inactive 125
  - types of 119
- sockets and ECBs 119
- SOCKSWP value, choosing 125
- SSL API functions
  - SSL\_accept 274
  - SSL\_aor 275
  - SSL\_check\_private\_key 277
  - SSL\_connect 279
  - SSL\_CTX\_check\_private\_key 280
  - SSL\_CTX\_free 281
  - SSL\_CTX\_load\_and\_set\_client\_CA\_list 282
  - SSL\_CTX\_load\_verify\_locations 283
  - SSL\_CTX\_new\_shared 287
  - SSL\_CTX\_new 285
  - SSL\_CTX\_set\_cipher\_list 289
  - SSL\_CTX\_set\_client\_CA\_list 292
  - SSL\_CTX\_set\_default\_passwd\_cb\_userdata 293
  - SSL\_CTX\_set\_verify 294
  - SSL\_CTX\_use\_certificate\_chain\_file 296
  - SSL\_CTX\_use\_certificate\_file 298
  - SSL\_CTX\_use\_PrivateKey\_file 300
  - SSL\_CTX\_use\_RSAPrivateKey\_file 302
  - SSL\_free 304
  - SSL\_get\_cipher 305
  - SSL\_get\_error 307
  - SSL\_get\_peer\_certificate 309
  - SSL\_get\_session 310
  - SSL\_get\_verify\_result 311
  - SSL\_get\_version 313
  - SSL\_library\_init 314
  - SSL\_load\_and\_set\_client\_CA\_list 315
  - SSL\_load\_client\_CA\_file 316
  - SSL\_new 317
  - SSL\_pending 318
  - SSL\_read 319
  - SSL\_renegotiate 320
  - SSL\_set\_cipher\_list 321
  - SSL\_set\_client\_CA\_list 324
  - SSL\_set\_fd 325
  - SSL\_set\_session 326
  - SSL\_set\_verify 327
  - SSL\_shutdown 329
  - SSL\_use\_certificate\_file 330
  - SSL\_use\_PrivateKey\_file 332
  - SSL\_use\_RSAPrivateKey\_file 333
  - SSL\_write 334
  - SSLv2\_client\_method 335
  - SSLv2\_server\_method 336
  - SSLv23\_client\_method 337
  - SSLv23\_server\_method 338
  - SSLv3\_client\_method 339
  - SSLv3\_server\_method 340
  - TLSv1\_client\_method 341
- SSL API functions (*continued*)
  - TLSv1\_server\_method 342
  - standard dotted decimal formats 120
  - start a CLAW logical link request 390
  - start an SSL session
    - SSL API function, SSL\_connect 279
  - starting an Internet server application 238
  - starting and stopping the IP trace function 75
  - starting the Internet daemon 237
  - starting the IP trace facility 407
  - STATE parameter 238
  - static VIPA
    - defined 64
  - stop all CLAW activity 395
  - stop CLAW activity on subchannel pair 388
  - stopping an Internet server application 238
  - stopping the Internet daemon 237
  - stopping the IP trace facility 408
  - storing data in the IP trace table 408
  - stream sockets
    - Transmission Control Protocol (TCP) 119
  - subchannel pair, stop activity 388
  - swinging VIPAs 68
  - syslog daemon 245
    - adding entry to IDCF 249
    - configuration file 246
    - diagnosing configuration problems 251
    - files used by 245, 246
    - offloading log files 251
    - operating 250
    - overview 245
    - starting 250
    - stopping 250
  - syslog daemon configuration file
    - example of 248
    - modifying 249
    - syntax 246
  - syslog.conf 246

## T

- table of MIB variables 423
- table, IP trace 407
- take away a logical link from an adapter 393
- TCP session, sample
  - function calls 127
- TCP, sample client code 373
- TCP, sample server code 371
- TCP/IP
  - error codes, socket 365
- TCP/IP commands
  - ZCLAW ACTIVATE 30
  - ZCLAW ADD 29
  - ZCLAW DELETE 31
  - ZCLAW DISPLAY 31
  - ZCLAW INACTIVATE 30
  - ZCLAW RESET 33
  - ZCLAW TRACE 32, 33
  - ZOSAE 63, 64, 65, 67, 68
  - ZTTCP DISPLAY 67
  - ZVIPA 65, 68, 69, 70, 71

- TCP/IP components
  - socket/CLAW interfaces 16
- TCP/IP layers 39
- TCP/IP native stack support
  - activate\_on\_accept API 82
  - activating and deactivating CDLC IP routers 65
  - activating and deactivating OSA-Express connections 67
  - balancing workloads with movable VIPAs 69
  - CDLC IP CCW area table 52
  - CDLC IP configuration record 51
  - components of 44
  - configuring a TPF system 59
  - data flow 41
  - deactivating sockets 77
  - default local IP address 61
  - defining
    - CDLC IP CCW area table resources 52
    - CDLC IP network configurations 52
    - CDLC IP routers 65
    - CDLC local IP addresses 62
    - gateways 58
    - IP message table 54
    - IP routing table 54
    - local IP addresses 60
    - OSA read buffers 53
    - OSA-Express cards 66
    - OSA-Express connections 67
    - real OSA IP addresses 63
    - routing table entries for SNMP 99
    - SNMP agent server 98
    - socket block table 51
    - TPF system to SNMP 99
    - VIPAs 64
  - deleting
    - CDLC IP routers 66
    - CDLC local IP addresses 62
    - OSA-Express connections 67
    - real OSA IP addresses 63
    - VIPAs 65
  - denial-of-service attacks 105
  - differentiated services 111
  - displaying
    - individual IP trace tables 77
    - IP trace information 75
    - OSA-Express connections 67
    - socket control block information 78
    - TCP/IP native stack support 75
    - VIPA statistics 69
    - VIPAs 65
  - enabling TCP/IP native stack support 60
  - inbound message flow 48
  - installation with SNMP 96
  - internals 39
  - Internet security 105
  - IP routers, configuration characteristics 41
  - IP routing table 54
  - local sockets 82
  - low-water marks 81
  - maximum packet size 61
  - new sock options supported 80
- TCP/IP native stack support (*continued*)
  - operator procedures 59
  - OSA-Express card, configuration characteristics 43
  - outbound message flow 46
  - packet filtering 105
  - performance 56
  - policy agent 46
  - restricted CDLC IP address 63
  - sample networks 41
  - select TCP/IP support user exit 124
  - send and receive buffer sizes 80
  - sharing sockets 79
  - SNMP agent support 85
  - socket application design considerations 79
  - socket block table structure 51
  - socket cycle-up 124
  - starting and stopping the IP trace function 75
  - TCP/IP layers 39
  - TCP/IP native stack support accept connection user exit 124
  - TCP/IP network services database support 111
  - timeouts 81
  - TPF control block structures 51
  - tuning major control block structures 55
  - tuning the IP network 55
  - tuning the IP over CDLC link layer 55
  - using existing sockets applications 79
  - using individual IP trace support 59
  - with individual IP trace support 76
  - workload balancing with movable VIPAs 69
- TCP/IP network
  - communication 11
  - configuration 11
  - description of a sample 13
  - offload device 11
    - TCP/IP offload device 11
- TCP/IP network services database
  - /etc/services file 113
  - data collection and reduction 112
  - differentiated services 111
  - message counters 112
  - network priority 111
  - network services database file 113
  - overview 111
  - quality of service (QoS) 111
  - type of service (TOS) 111
- TCP/IP operator procedures
  - activating CLAW workstations 30
  - changing hardware 32
  - configuring a TPF system 29
  - data trace 32
  - deactivating CLAW workstations 30
  - defining CLAW workstations 29
  - defining the CLAW host name 29
  - deleting a CLAW workstation 31
  - displaying TCP/IP information 31
  - moving a CLAW workstation 31
  - process trace 33
  - resetting message lock 33
- TCP/IP restricted C functions
  - CLAW functions 385

- TCP/IP restricted C functions (*continued*)
  - reference 385
- TCP/IP restricted C functions: reference 385
- TCP/IP, workstation interface
  - restricted C functions 385
  - TPF, used in 385
- terminate all CLAW activity 395
- terminate CLAW activity on subchannel pair 388
- TFTP configuration file 241
  - creating 241
  - directives 240
  - general discussion 239
  - transferring to TPF system 241
  - updating 241
- TFTP server 239
- tftp.conf 241
- timeouts 81
- TPF configuration file
  - /etc/tpf\_mail.conf 259
  - creating 259
  - general discussion 259
  - parameters 263
- TPF host name table 101
- TPF Internet mail server
  - #MAILxx record 255, 258
  - access control list (ACL) 256
  - access list 259
  - active queue 257
  - active queue control record 257
  - active queue record 257
  - adding an entry to IDCF 267
  - adding new users 268
  - administrator tasks 265
  - client tasks 269
  - configuration files 259
  - configuring 265
  - controlling 268
  - database layout 255
  - deferred queue 257
  - deferred queue control record 257
  - deferred queue record 257
  - IMAP 253
  - mail queue 257
  - managing mailboxes 269
  - operator tasks 265
  - overview 253
  - POP 253
  - receiving mail 269
  - recoup considerations 258
  - sending mail 269
  - SMTP 253
  - updating 267
  - user profile record (UPR) 255
- trace postprocessor
  - CLAW process trace output 350
  - CLAW sample trace output, data 344
- trace postprocessor, CLAW data 343
- trace postprocessor, CLAW process 350
- trace postprocessor, data
  - JCL, sample 343
- trace postprocessor, process
  - JCL, sample 350
- transferring the TFTP configuration file to the TPF system 241
- translate a long integer
  - socket API function, htonl 180
  - socket API function, ntohl 191
- translate a short integer
  - socket API function, htons 181
  - socket API function, ntohs 192
- Transmission Control Protocol (TCP)
  - stream sockets 119
- transmission, data
  - socket concepts used 120
- traps
  - and SNMP 93
- Trivial File Transfer Protocol (TFTP) server
  - adding entry to IDCF 239
  - general discussion 239
- tuning TCP/IP native stack support
  - IP network 55
  - IP over CDLC link layer 55
  - major control block structures 55
- tuning, TCP/IP native stack support 54
- type of service (TOS) 111

## U

- UCOM user exit with SNMP
  - and variable binding list 91
  - coding 98
  - message processing 91
- UDP session, sample
  - function calls 131
- UDP, sample client code 378
- UDP, sample server code 376
- UMIB user exit with SNMP
  - coding 98
  - message processing 91
- updating
  - Internet daemon configuration file 235
  - TFTP configuration file 241
- used in data transmission
  - socket concepts 120
- User Datagram Protocol (UDP)
  - datagram sockets 119
- user exit and movable VIPAs 69
- user exits, nonsocket
  - system states for 20
- user exits, socket
  - cycle-up 123
  - select TCP/IP support user exit 124
  - socket accept 123
  - socket activation 123
  - socket connect 123
  - socket cycle-up user exit 124
  - socket deactivation 123
  - system error 124
  - TCP/IP native stack support accept connection user exit 124
- user MIB variables and SNMP 93

USER parameter 237  
user profile record (UPR) 255  
using existing sockets applications 79  
UVIP user exit and movable VIPAs 69

ZTTCP DELETE command 62, 66  
ZTTCP DISPLAY command 67  
ZTTCP INACTIVATE command 65, 67  
ZVIPA command 65, 68, 69, 70, 71

## V

variable binding lists, SNMP 91  
variables table, MIB 423  
verify private key- public key agreement in the certificate  
  SSL API function, SSL\_CTX\_check\_private\_key 280  
verify private key-public key agreement  
  SSL API function, SSL\_check\_private\_key 277  
VIPAC macro use for moving a VIPA 69  
virtual IP addresses (VIPAs)  
  and workload balancing 69  
  defining 64  
  deleting 65  
  displaying 65  
  movable 64  
  moving by application program 69  
  moving to another processor 68  
  static 64  
  swinging 68  
  types 64  
  using with an OSA-Express connection 63

## W

WAN 3  
workload balancing with movable VIPAs 69  
write application data across an SSL session  
  SSL API function, SSL\_write 334  
write data on a connected socket  
  socket API function, write 226  
  socket API function, writev 229  
writing the IP trace table to a real-time tape 409

## Z

ZCACHE command 103  
ZCLAW ACTIVATE 30  
ZCLAW ADD 29  
ZCLAW DELETE 31  
ZCLAW DISPLAY 31  
ZCLAW INACTIVATE 30  
ZCLAW RESET 33  
ZCLAW TRACE 32, 33  
ZFILE kill command 250  
ZFILE rm command 243  
ZINET ADD command 99, 237, 249  
ZINET ALTER command 235, 237  
ZINET DELETE command 235  
ZINET START command 99, 237, 238, 250  
ZINET STOP command 237, 238, 250  
ZOSAE command 63, 64, 65, 67, 68  
ZSNMP command 98  
ZTRTE command 99  
ZTTCP ACTIVATE command 65, 67  
ZTTCP CHANGE command 62, 63  
ZTTCP DEFINE command 62, 63, 65







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.

SH31-0120-12

